

Lời nói đầu

Các mẫu thiết kế (Design Patterns) được xem như phần tiếp theo của quá trình học lập trình Hướng đối tượng. Các mẫu thiết kế là kết quả rút ra từ thực tiễn lập trình, vì vậy mang đến ý nghĩa thực hành rất lớn cho học viên học lập trình Hướng đối tượng. Trong tài liệu này, mình giới thiệu 23 mẫu thiết kế được định nghĩa trong cuốn sách kinh điển "Design Patterns - Elements of Reusable Object-Oriented Software" [Addison-Wesley, 1995] của Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides (GoF – Gang of Four).

Mình cố gắng trình bày một cách đơn giản, trong sáng, giúp bạn trong âu lạc bộ [UPC- UTT Programming Club](#) nắm bắt được tinh thần của các mẫu thiết kế. Các ví dụ và bài tập (bằng Java) cũng được cân nhắc lựa chọn nhằm làm nổi bật đặc điểm lập trình của các mẫu thiết kế.

Mình là Hoàng Anh Tiến một thành viên của [UPC- UTT Programming Club](#) đặc biệt cảm ơn tới câu lạc bộ đã cho mình động lực viết giáo trình này.

Mặc dù đã dành rất nhiều thời gian và công sức cho tài liệu này, phải hiệu chỉnh chi tiết và nhiều lần, nhưng tài liệu không thể nào tránh được những sai sót và hạn chế. Mình thật sự mong nhận được các ý kiến góp ý từ bạn đọc để tài liệu có thể hoàn thiện hơn.

Các bạn trong câu lạc bộ nếu có sử dụng giáo trình này, xin gửi cho mình ý kiến đóng góp phản hồi, giúp giáo trình được hoàn thiện thêm, phục vụ cho công tác giảng dạy chung và góp phần giúp câu lạc bộ phát triển và lớn mạnh hơn.

Phiên bản

Cập nhật ngày: 06/09/2024

Thông tin liên lạc

Mọi ý kiến và câu hỏi có liên quan xin vui lòng gửi về:

+ Địa chỉ: 54, Triều Khúc, P. Thanh Xuân Nam, Q. Thanh Xuân, Hà Nội

+ Sdt: 033 228 4267

+ Fanpage: [UPC- UTT Programming Club](#)

+ Email: uttprogrammingclub@gmail.com

GoF Design Patterns

Design patterns là tập hợp các mẫu thiết kế, dùng như giải pháp thực tế hoặc như thiết kế chuẩn cho các vấn đề phổ biến trong xây dựng phần mềm hướng đối tượng. Được áp dụng trong toàn bộ vòng đời phát triển phần mềm, các mẫu thiết kế giúp ứng dụng có tổ chức tốt, linh hoạt, dễ bảo trì và nâng cấp.

Tài liệu này giới thiệu 23 mẫu thiết kế được định nghĩa trong cuốn sách kinh điển "Design Patterns - Elements of Reusable Object-Oriented Software" [Addison-Wesley, 1995] của Erich Gamma, Richard Helm, Ralph Johnson, và John Vlissides (GoF – Gang of Four) [1].

GoF phân loại các mẫu thiết kế theo hai cách:

- Mục đích của mẫu thiết kế: có ba nhóm.

Creational gồm 5 mẫu thiết kế: Abstract Factory, Builder, Factory Method, Prototype và Singleton. Nhóm này mô tả việc trừu tượng hóa quá trình tạo ra các thể hiện của đối tượng, thay vì khởi tạo trực tiếp từ constructor. Lưu ý, tính đa hình không làm việc khi chúng ta tạo đối tượng.

Structural gồm 7 mẫu thiết kế: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Nhóm này mô tả cách phối hợp các lớp và các đối tượng để hình thành một cấu trúc phức tạp hơn. Nhóm này được sử dụng khi thiết kế hệ thống mới hoặc khi bảo trì, mở rộng hệ thống có sẵn.

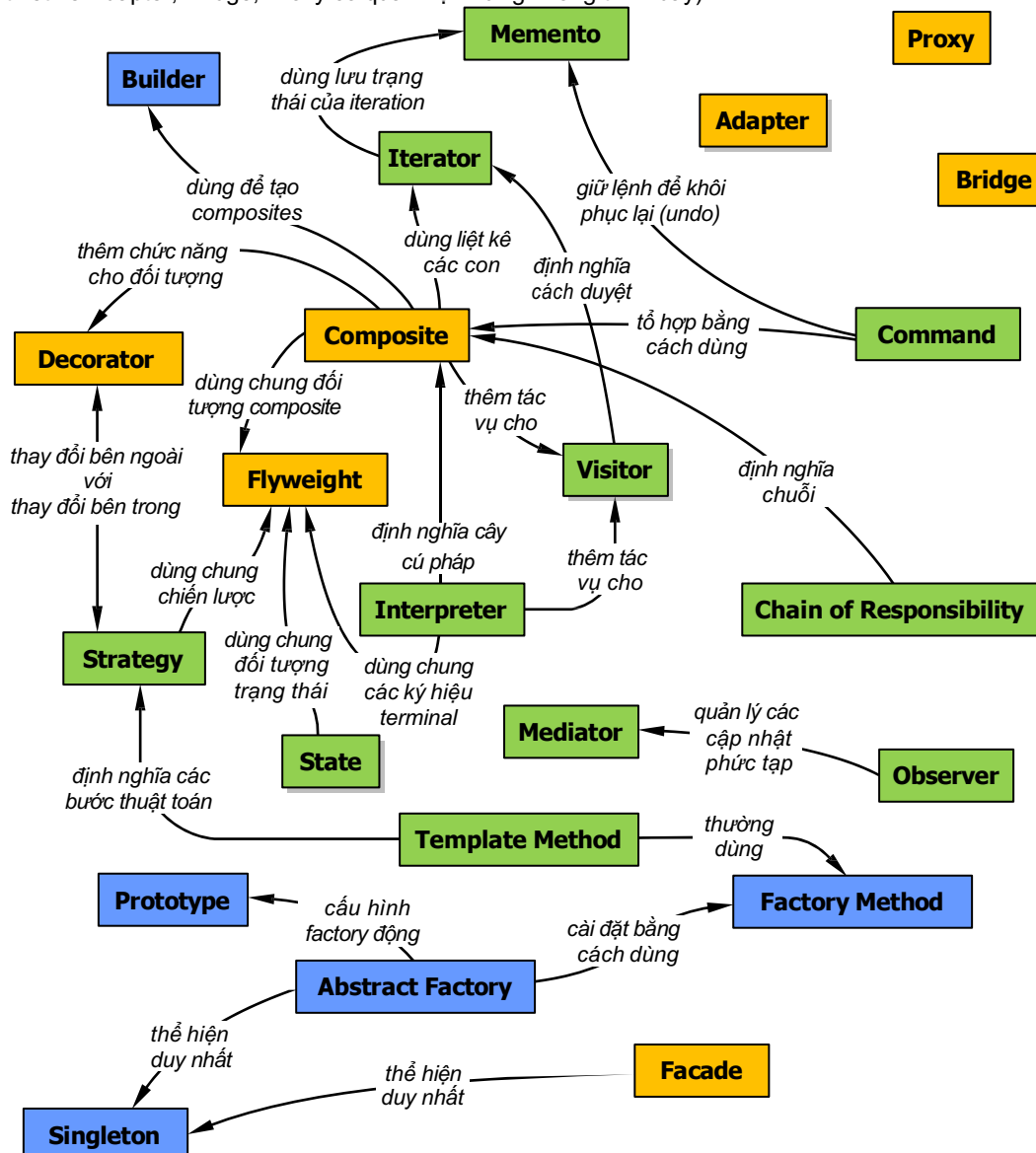
Behavioral gồm 11 mẫu thiết kế: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method và Visitor. Nhóm này quan tâm đến việc trao đổi thông tin, phân chia trách nhiệm, tương tác giữa các đối tượng.

- Phạm vi (scope) quan hệ: có hai nhóm.

Class Factory Method, Adapter (class), Interpreter và Template Method.

Object Abstract Factory, Builder, Prototype, Singleton, Adapter (object), Bridge, Composite, Decorator, Facade, Flyweight, Proxy, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor.

Trong thực tế, các mẫu thiết kế thường được dùng phối hợp với nhau. GoF trình bày quan hệ giữa các mẫu thiết kế như hình sau (các mẫu thiết kế Adapter, Bridge, Proxy có quan hệ nhưng không trình bày).



Quan hệ giữa các mẫu thiết kế [Gamma et al. 1995]

Tham khảo sơ đồ quan hệ này sau khi đã hiểu rõ 23 mẫu thiết kế

Steven John Metsker [7] lại phân loại các mẫu thiết kế thành 5 nhóm:

Interface gồm Adapter, Facade, Composite và Bridge. Giải quyết các vấn đề liên quan đến giao diện.

Responsibility gồm Singleton, Observer, Mediator, Proxy, Chain of Responsibility và Flyweight. Giải quyết các vấn đề liên quan đến phân công trách nhiệm cho đối tượng.

Construction gồm Builder, Factory Method, Abstract Factory, Prototype và Memento. Giải quyết các vấn đề về tạo đối tượng mà không dùng constructor.

Operation gồm Template Method, State, Strategy, Command và Interpreter. Giải quyết các vấn đề điều khiển tác vụ, xử lý khi có nhiều tác vụ tham gia.

Extension gồm Decorator, Iterator và Visitor. Giải quyết vấn đề mở rộng chức năng.

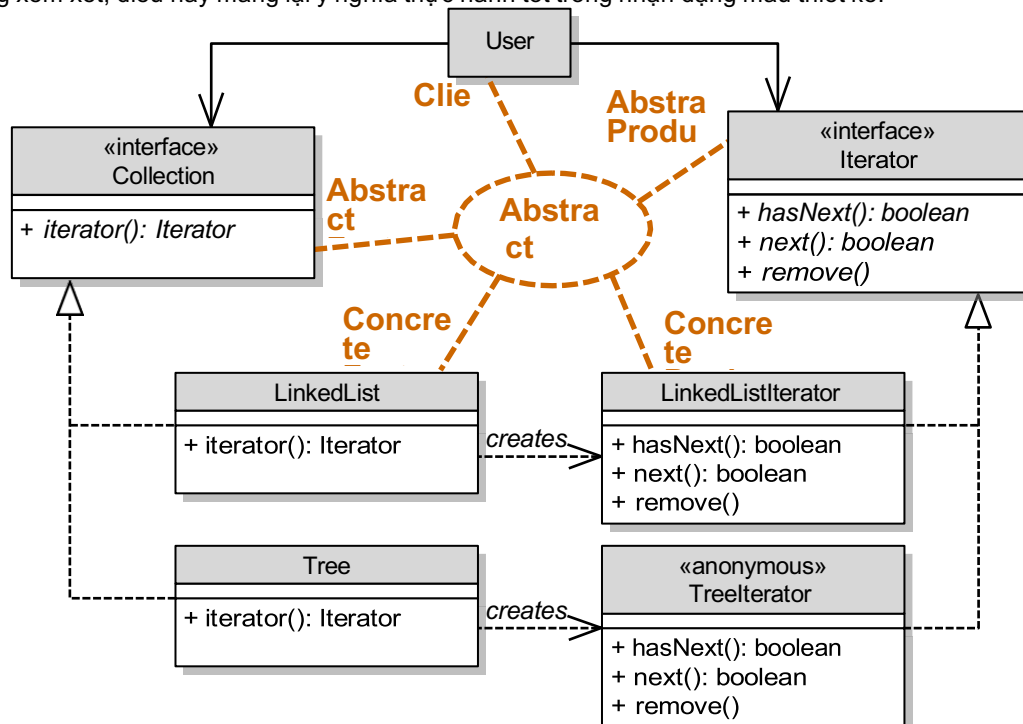
Một mẫu thiết kế ánh xạ giữa một vấn đề thường gặp phải khi thiết kế và một giải pháp chung:

- Đã tạo trực tiếp các đối tượng có nhiệm vụ rõ ràng: hệ thống gắn liền với cài đặt quá cụ thể sẽ làm mất đi tính linh động, khó mở rộng. Giải pháp: tạo các đối tượng một cách gián tiếp bằng cách dùng Abstract Factory, Factory Method, Prototype.
- Phụ thuộc vào các tác vụ cụ thể: hệ thống chỉ có một cách để đáp ứng yêu cầu. Giải pháp: tránh viết code cố định bằng cách dùng Chain of Responsibility, Command.
- Phụ thuộc vào nền tảng phần cứng hoặc phần mềm: ứng dụng khó chuyển đến các nền tảng khác. Giải pháp: giới hạn sự phụ thuộc nền tảng bằng cách dùng Abstract Factory và Bridge.
- Phụ thuộc vào giao diện người dùng hoặc code của Client: giao diện người dùng và code của Client có thể bị thay đổi nếu các đối tượng trong hệ thống thay đổi. Giải pháp: cách ly với Client, bằng cách dùng Abstract Factory, Bridge, Memento, Proxy.
- Phụ thuộc vào thuật toán: thuật toán sử dụng thay đổi thường xuyên. Khi thuật toán thay đổi, các đối tượng phụ thuộc nó buộc phải thay đổi. Giải pháp: cô lập thuật toán, dùng Builder, Iterator, Strategy, Template Method, Visitor.
- Ràng buộc quá chặt chẽ: các lớp liên kết với nhau quá chặt chẽ sẽ rất khó sử dụng lại, khó kiểm tra, bảo trì. Giải pháp: làm suy yếu liên kết quá chặt chẽ giữa các lớp bằng cách dùng Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.
- Mở rộng chức năng của lớp bằng thừa kế: thừa kế (inheritance) khó sử dụng, khó hiểu hơn tổng hợp (composition). Giải pháp: mở rộng chức năng tránh dùng thừa kế mà dùng tổng hợp, với Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.
- Không dễ dàng tùy biến các lớp: các lớp không thể tiếp cận, không thể hiểu hoặc khó thay đổi. Giải pháp: không can thiệp vào lớp mà mở rộng bằng cách dùng Adapter, Decorator, Visitor.

Học các mẫu thiết kế là học kinh nghiệm từ thực tiễn, nâng cao tư duy thiết kế hướng đối tượng, nắm bắt nguyên tắc cấu trúc ứng dụng, biết cách tổ chức lại (refactoring) ứng dụng. Tuy nhiên, do bạn hoàn toàn có thể xây dựng chương trình không dùng các mẫu thiết kế, bạn cần nhận thức được lợi ích khi học và sử dụng các mẫu thiết kế. Cách tiếp cận như sau:

- Trước tiên, bạn chấp nhận giả thuyết cho rằng các mẫu thiết kế là quan trọng trong việc thiết kế hệ thống phần mềm.
- Thứ hai, bạn phải nhận ra rằng bạn cần phải đọc về các mẫu thiết kế để biết khi nào bạn có thể sử dụng chúng.
- Thứ ba, bạn phải hiểu các mẫu thiết kế một cách chi tiết đủ để biết loại nào trong chúng có thể giúp bạn giải quyết yêu cầu thiết kế hoặc vấn đề gặp phải khi thiết kế.

Bạn nên học tập cách ghi chú trực quan của Allen Holub [4] về mối tương quan giữa các thành phần của mẫu thiết kế với hệ thống lớp đang xem xét, điều này mang lại ý nghĩa thực hành tốt trong nhận dạng mẫu thiết kế.



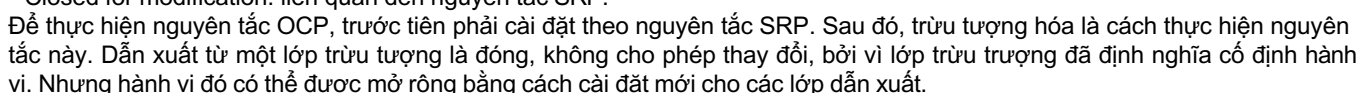
Do kết quả thiết kế thường là một mô hình giải pháp trừu tượng biểu diễn bằng ngôn ngữ UML. Bạn có thể dùng UMLet, tại: <http://www.umlet.com> để vẽ sơ đồ lớp. Với các ví dụ trong tài liệu, bạn nên dùng ObjectAid UML Explorer, plugin của Eclipse, tại: <http://www.objectaid.net>. ObjectAid UML Explorer cho phép chuyển ngược các lớp Java trở lại thành sơ đồ lớp.

Các mẫu thiết kế trong tài liệu được trình bày theo thứ tự giống sách của GoF [1]. Theo chúng tôi, để dễ tiếp cận, bạn nên tìm hiểu theo thứ tự sau: Singleton, Iterator, Adapter, Decorator, State, Strategy, Factory Method, Observer, Facade, Template Method, rồi tự lựa chọn thứ tự tìm hiểu các mẫu thiết kế còn lại.

- Don't Repeat Yourself (DRY), không nên viết những đoạn code trùng lặp trong cùng một chương trình (low duplicate) mà nên dùng lớp trừu tượng hoặc viết thành phương thức để dùng chung.
- Giữ liên kết giữa các lớp không quá chặt (loosely coupling).
- Đóng gói những gì dễ thay đổi.
- Nên dùng tổng hợp (composition) hơn là dùng thừa kế (inheritance).
- Client luôn làm việc với phần trừu tượng (sử dụng polymorphism), không trực tiếp làm việc với cài đặt.
- Ủy nhiệm (delegation) cho đối tượng tự thực hiện hành vi mong muốn, thay vì thực hiện hành vi đó lên đối tượng.

- **S**ingle Responsibility
- **O**pen-Closed
- **L**iskov's Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Ví dụ, lớp Employee sau đảm nhiệm cùng lúc ba nhiệm vụ, vi phạm nguyên tắc SRP:



Vi phạm nguyên tắc OCP	Đảm bảo nguyên tắc OCP
<pre> class Square { double side; } class Circle { double radius; } class AreaCalculator { public double Area(Object[] shapes) { double area = 0; for (Object object : shapes) { if (object instanceof Square) { Square square = (Square) object; area += square.side * square.side; } if (object instanceof Circle) { Circle circle = (Circle) object; area += circle.radius * circle.radius * Math.PI; } } return area; } } </pre>	<pre> abstract class Shape { public abstract double area(); } class Square extends Shape { double side; @Override public double area() { return side * side; } } class Circle extends Shape { double radius; @Override public double area() { return radius * radius * Math.PI; } } class AreaCalculator { public double Area(Shape[] shapes) { double area = 0; for (Shape shape : shapes) { area += shape.area(); } return area; } } </pre>

Trong thiết kế bên trái, lớp AreaCalculator sử dụng trực tiếp các lớp Square và Circle, nên vi phạm nguyên tắc OCP. Nếu mở rộng, ví dụ thêm lớp Triangle (tam giác), buộc phải thay đổi nhiều trong lớp AreaCalculator.

Liskov's Substitution (LSP)

[Barbara Liskov] Kiểu dẫn xuất phải có khả năng thay thế được kiểu cơ sở của nó. Nói một cách đơn giản, lớp dẫn xuất phải được cài đặt sao cho nó không phá vỡ chức năng của lớp cha dưới góc nhìn của người dùng. Người dùng sử dụng một lớp, và họ mong tiếp tục làm việc đúng với lớp dẫn xuất từ lớp đó. LSP thường vi phạm khi bạn loại bỏ một số tính năng của lớp.

Vi phạm nguyên tắc LSP	Đảm bảo nguyên tắc LSP
<pre> abstract class Teacher { int name; public abstract void teach(); private void takeAttendance() { System.out.println("Take attendance"); } private void collectPaper() { System.out.println("Collect papers"); } public void performOtherTasks() { takeAttendance(); collectPaper(); } } class MathTeacher extends Teacher { @Override public void teach() { System.out.println("Calculate math"); } } class SubstitutionTeacher extends Teacher { @Override public void teach() { // cannot teach? throw exception? } } </pre>	<pre> class Staff { int name; private void takeAttendance() { System.out.println("Take attendance"); } private void collectPaper() { System.out.println("Collect papers"); } public void performOtherTasks() { takeAttendance(); collectPaper(); } } interface Instructor { void teach(); } class MathTeacher extends Staff implements Instructor { @Override public void teach() { System.out.println("Calculate math"); } } class SubstitutionTeacher extends Staff { </pre>

Thiết kế bên trái vi phạm nguyên tắc LSP. Lớp SubstitutionTeacher, dẫn xuất từ Teacher, hoạt động không như mong đợi. Thực tế, nguyên nhân do lớp SubstitutionTeacher không phải là Teacher, nó không cần đến phương thức teach().

Interface Segregation (ISP)

[Robert C. Martin] Giao diện lớn nên tách thành nhóm các giao diện có chức năng đặc thù hơn, mỗi giao diện thu hẹp như vậy gọi là giao diện vai trò (role interface), phục vụ cho một tập khách hàng riêng.

Thiết kế theo ISP giữ cho hệ thống tách biệt, dễ dàng tổ chức lại, thay đổi hoặc tái bố trí.

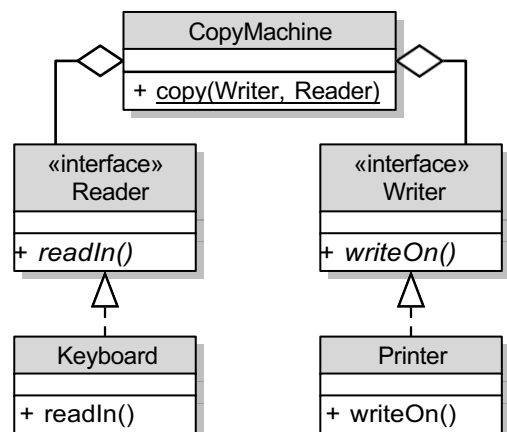
Vi phạm nguyên tắc ISP	Đảm bảo nguyên tắc ISP
<pre> interface IWorker { void work(); void eat(); } class Worker implements IWorker { @Override public void work() { System.out.println("worker working"); } @Override public void eat() { System.out.println("eating"); } } class Robot implements IWorker { @Override public void work() { System.out.println("robot working"); } @Override public void eat() { System.out.println("no need eat"); } } public class Manager_noISP { public static void main(String[] args) { IWorker[] list = {new Worker(), new Robot()}; for (IWorker worker : list) worker.work(); } } </pre>	<pre> interface IWorkable { void work(); } interface IFeedable { void eat(); } interface IWorker extends IFeedable, IWorkable { } class Worker implements IWorker { @Override public void work() { System.out.println("worker working"); } @Override public void eat() { System.out.println("eating"); } } class Robot implements IWorkable { @Override public void work() { System.out.println("robot working not eating"); } } public class Manager_ISP { public static void main(String[] args) { IWorkable[] list = {new Worker(), new Robot()}; for (IWorkable worker : list) worker.work(); } } </pre>

Dependency Inversion (DIP)

[Robert C. Martin] Ngược với kiến trúc truyền thống, những module ở mức cao (nơi phối hợp hoạt động của nhiều module ở mức thấp) không nên phụ thuộc trực tiếp vào những module mức thấp (nơi thực hiện chức năng cơ bản), cả hai nên phụ thuộc thông qua lớp trừu tượng.

Kiến trúc: thay thế (high-module → low-module) bằng (high-module → [interface ← low-module])

<pre> interface Writer { void writeOn(); } interface Reader { void readIn(); } class Keyboard implements Reader { @Override public void readIn() { System.out.println("Read in keyboard"); } } class Printer implements Writer { @Override public void writeOn() { System.out.println("Write on paper"); } } </pre>	<pre> public class CopyMachine { public static void copy(Writer w, Reader r) { r.ReadIn(); w.writeOn(); } public static void main(String[] args) { Writer w = new Printer(); Reader r = new Keyboard(); copy(w, r); } } </pre>
--	---



Kiến trúc hình bên giúp module mức cao (copy()) không bị ảnh hưởng khi thay đổi các module mức thấp (thay đổi loại Reader hoặc Writer).

Creational

Abstract Factory

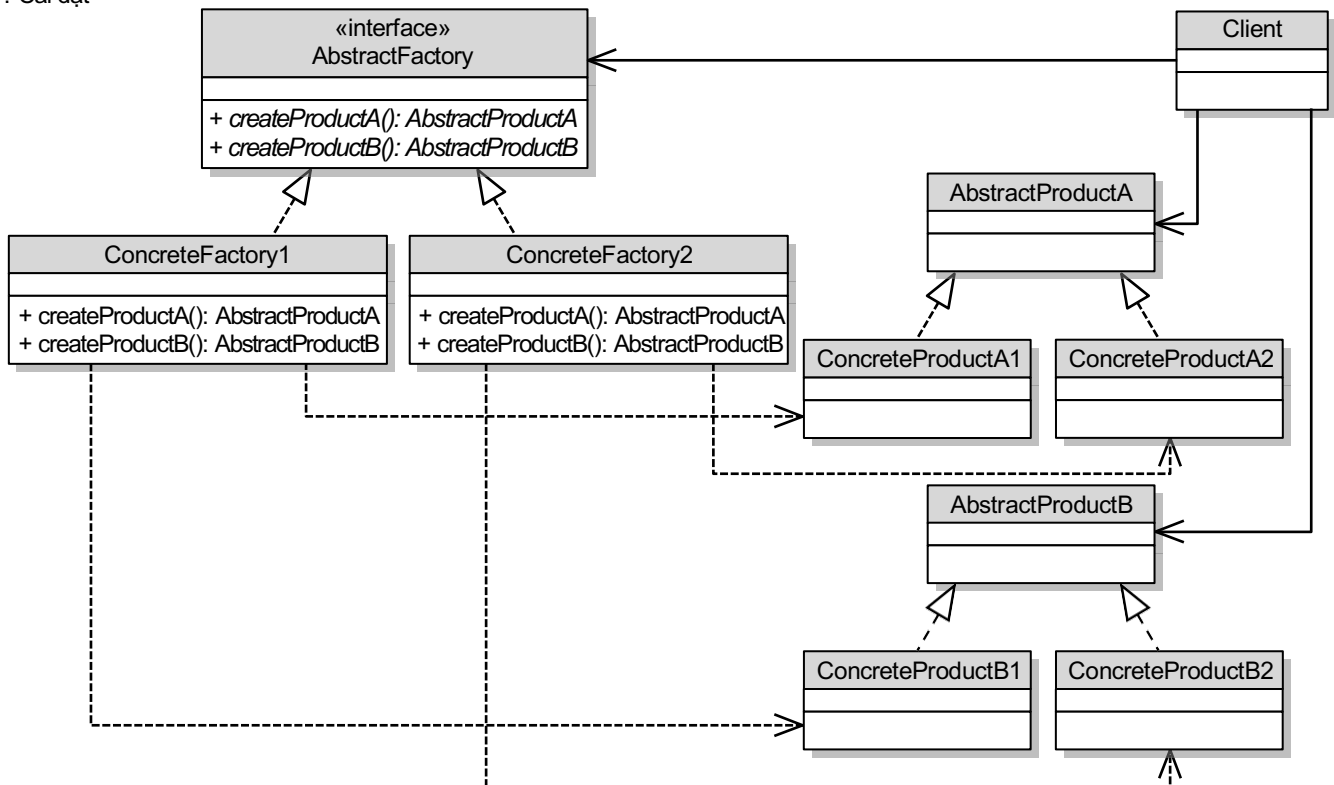
Create object families 

Mẫu thiết kế Abstract Factory cung cấp một giao diện dùng để tạo ra một họ các đối tượng có quan hệ (hoặc phụ thuộc) với nhau mà không cần chỉ định rõ lớp cụ thể của chúng. Họ đối tượng (object family) không mang ý nghĩa chúng cùng cây thừa kế, mà là các đối tượng có quan hệ (hoặc phụ thuộc) với nhau, mỗi họ được sử dụng trong một ngữ cảnh khác nhau. Ví dụ:

- Ngữ cảnh là theme sử dụng: window và scrollbar có theme màu vàng thuộc họ đối tượng YellowThemeWidget; họ này được tạo ra từ factory cụ thể, YellowThemeWidgetFactory.
- Ngữ cảnh là hệ nền chạy chương trình: window và scrollbar chạy trên Unix thuộc họ đối tượng XWindowsComponent; họ này được tạo ra từ factory cụ thể, XWindowsFactory.

Tập hợp các phương thức dùng tạo đối tượng được đóng gói trong một đối tượng tạo, gọi là đối tượng factory.

1. Cài đặt



- AbstractFactory: khai báo giao diện chứa một tập hợp các phương thức tạo các loại đối tượng.
- ConcreteFactory: cài đặt cụ thể các phương thức tạo đối tượng để tạo ra họ đối tượng.
- AbstractProduct: khai báo giao diện chung cho loại đối tượng sẽ được tạo. Ví dụ, window và scrollbar là hai loại đối tượng.
- ConcreteProduct: các đối tượng thật sự được tạo ra bởi factory. Ví dụ, window màu vàng và scrollbar màu hồng.
- Client: sử dụng các đối tượng được tạo ra thông qua giao diện. Do Client không xác định được ngữ cảnh tạo đối tượng nên phải dùng Abstract Factory.

```

// (1) AbstractProduct
abstract class Window {
    public abstract void draw();
}

abstract class Scrollbar {
    public abstract void paint();
}

// (2) ConcreteProduct
class YellowThemeWindow extends Window {
    @Override public void draw() {
        System.out.println(getClass().getName());
    }
}

class PinkThemeWindow extends Window {
    @Override public void draw() {
        System.out.println(getClass().getName());
    }
}
  
```

```

class YellowThemeScrollbar extends Scrollbar {
    @Override public void paint() {
        System.out.println(getClass().getName());
    }
}

class PinkThemeScrollbar extends Scrollbar {
    @Override public void paint() {
        System.out.println(getClass().getName());
    }
}

// (3) AbstractFactory
interface WidgetFactory {
    Scrollbar createScrollbar();
    Window createWindow();
}

// (4) ConcreteFactory
class YellowThemeWidgetFactory implements WidgetFactory {
    @Override public Scrollbar createScrollbar() {
        return new YellowThemeScrollbar();
    }

    @Override public Window createWindow() {
        return new YellowThemeWindow();
    }
}

class PinkThemeWidgetFactory implements WidgetFactory {
    @Override public Scrollbar createScrollbar() {
        return new PinkThemeScrollbar();
    }

    @Override public Window createWindow() {
        return new PinkThemeWindow();
    }
}

public class Client {
    public static void initGUI(WidgetFactory factory) {
        factory.createScrollbar().paint();
        factory.createWindow().draw();
    }

    public static void main(String[] args) {
        System.out.println("--- Abstract Factory Pattern ---");
        initGUI(new PinkThemeWidgetFactory());
    }
}

```

Một họ đối tượng là các widget (window, scrollbar) thiết kế đồ họa theo cùng một chủ đề (theme). Bằng cách thay đổi factory tạo họ đối tượng, bạn có thể thay đổi theme áp dụng cho chúng một cách dễ dàng.

2. Liên quan

- Prototype: lớp AbstractFactory thường được cài đặt bằng các Factory Method, nhưng nó cũng được cài đặt bằng Prototype.
- Singleton: lớp ConcreteFactory thường là một Singleton.

3. Java API

Nhiều lớp của JAXP (DocumentBuilderFactory, SAXParserFactory, TransformFactory, XPathFactory, ...) là AbstractFactory, chúng có phương thức tạo trả về một factory, factory này được dùng để tạo các kiểu abstract/interface khác.

Lớp java.awt.Toolkit là một Abstract Factory được dùng để tạo các đối tượng làm việc với hệ thống cửa sổ thuộc các hệ nền khác nhau.

Trong JavaEE, các mẫu thiết kế Data Access Object (DAO, lớp Integration) và Transfer Object Assembler (lớp Business) cũng áp dụng mẫu thiết kế Abstract Factory.

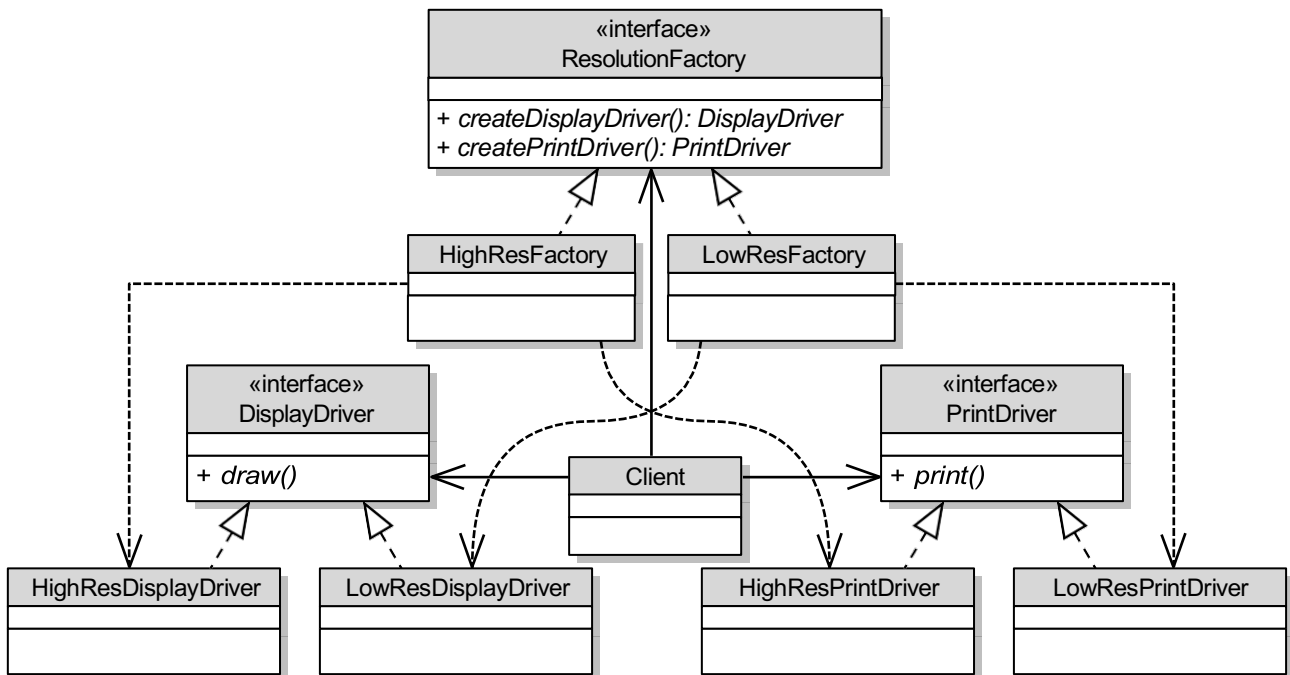
4. Sử dụng

Ta muốn:

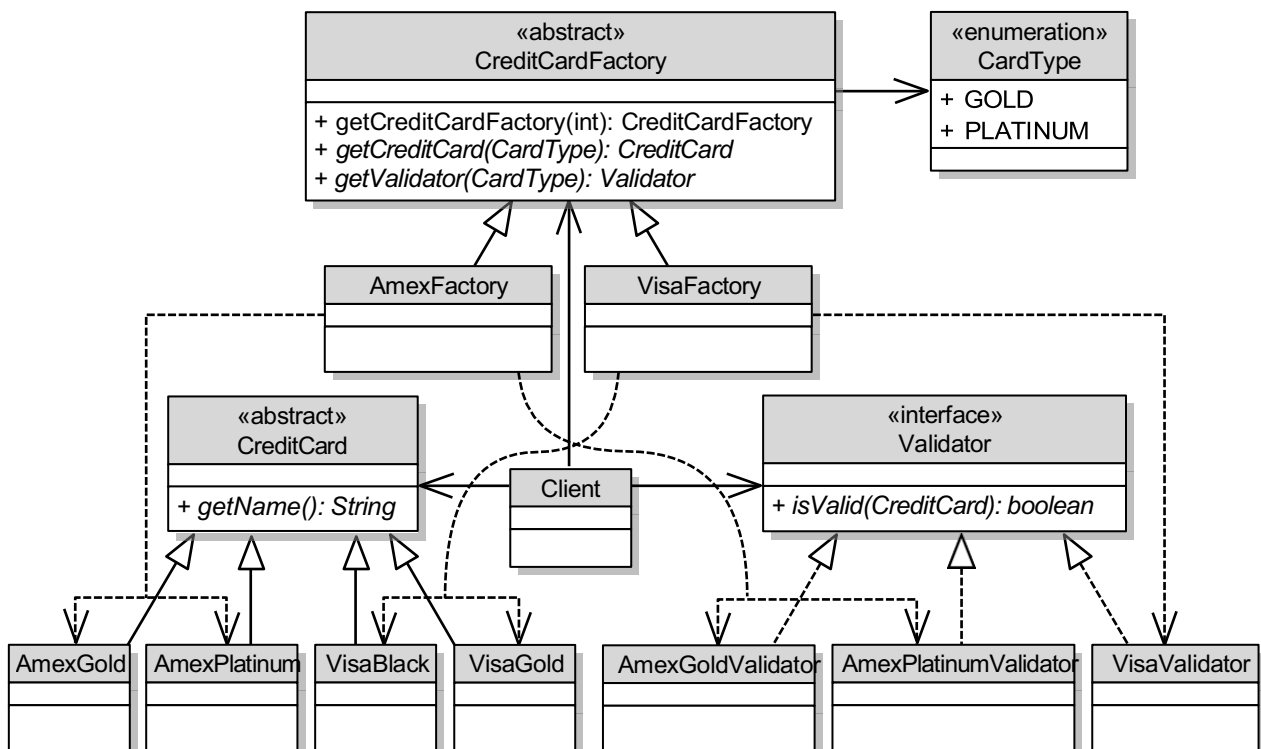
- Một hệ thống độc lập với cách mà các sản phẩm của nó được tạo ra, được tích hợp và được thể hiện.
- Một hệ thống được cấu hình để tạo ra một hoặc nhiều họ sản phẩm, hoạt động trong nhiều ngữ cảnh.
- Yêu cầu bắt buộc các sản phẩm trong một họ phải làm việc với nhau.
- Cung cấp một thư viện lớp các sản phẩm, nhưng chỉ muốn để cập đến phần giao diện, chưa chú ý đến phần cài đặt.

5. Bài tập

a) Driver cho màn hình và máy in trong thuộc hai họ driver: driver có độ phân giải thấp (low-resolution) và driver có độ phân giải cao (high-resolution). Bạn hãy áp dụng mẫu thiết kế Abstract Factory để tạo ra các họ driver thích hợp.



b) CreditCard có 2 loại: AmexCreditCard (gồm AmexGold và AmexPlatinum) và VisaCreditCard (VisaBlack và VisaGold). Hai loại AmexCreditCard được xác minh bằng Validator khác nhau (AmexGoldValidator và AmexPlatinumValidator), trong lúc hai loại VisaCreditCard đều được xác minh bằng một Validator chung (VisaValidator). Loại CreditCard được tạo tùy theo creditScore (creditScore > 500 cho AmexCreditCard và ngược lại cho VisaCreditCard) và CardType. Bạn hãy áp dụng mẫu thiết kế Abstract Factory để tạo ra các họ CreditCard và áp dụng loại Validator thích hợp.



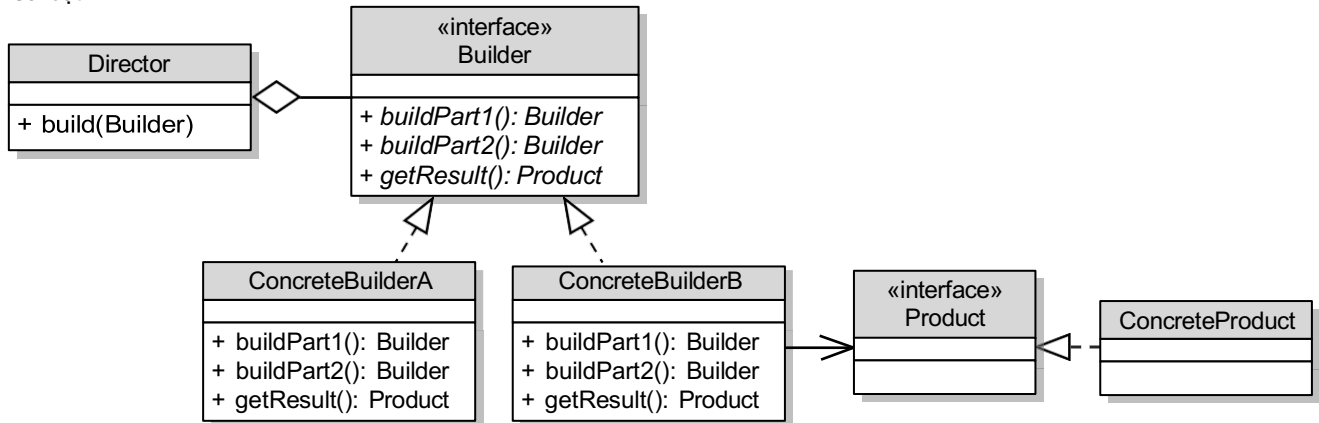
Builder

Building complex objects 

Mẫu thiết kế Builder tách việc khởi tạo nhiều bước của một đối tượng phức tạp (Product) thành tập hợp các thao tác khởi tạo từng phần của đối tượng đó. Lớp Director quyết định các bước khởi tạo một đối tượng phức tạp từ nhiều thành phần, trong lúc các lớp Builder thực sự xây dựng các thành phần đó.

Sự tách biệt trách nhiệm này cho phép cùng một quá trình khởi tạo có thể tạo ra các thể hiện khác nhau. Ngoài ra, đối tượng có thể khởi tạo từng phần trong trường hợp chưa có đầy đủ dữ liệu dùng cho khởi tạo.

1. Cài đặt



- Builder: cung cấp giao diện để Director tạo ra các thành phần của đối tượng phức tạp Product. Từ các thành phần này, Director sẽ xây dựng nên Product.
- ConcreteBuilder: cài đặt cụ thể cho các phương thức tạo ra các thành phần của Product. Thay đổi ConcreteBuilder khác cho ra Product khác.
- Director: Client gọi Director để tạo ra Product từ các thành phần do Builder cung cấp. Director được dùng để quyết định số lượng và thứ tự các bước xây dựng Product, nó gọi các Builder khi cần các thành phần để hình thành nên Product.

// (1) Builder

```
interface QueryBuilder {
    QueryBuilder setFrom(String from);
    QueryBuilder setWhere(String where);
    Query getQuery();
}
```

// (2) ConcreteBuilder

```
class SQLQueryBuilder implements QueryBuilder {
    private Query query = new SQLQuery();
    @Override public QueryBuilder setFrom(String from) {
        query.add(from);
        return this;
    }

    @Override public QueryBuilder setWhere(String where) {
        query.add(where);
        return this;
    }

    @Override public Query getQuery() {
        return query;
    }
}
```

// (3) Product

```
interface Query {
    void add(String s);
    void execute();
}
```

// (4) ConcreteProduct

```
class SQLQuery implements Query {
    StringBuilder query = new StringBuilder();
    @Override public void add(String part) {
        query.append(" ").append(part);
    }

    @Override public void execute() {
        System.out.printf("Execute \"%s\"%n", query.toString().trim());
    }
}
```

```

    }
}

// (5) Director
class QueryBuildDirector {
    public Query build(QueryBuilder builder, String from, String where) {
        return builder.setFrom(from).setWhere(where).getQuery();
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Builder Pattern ---");
        QueryBuildDirector director = new QueryBuildDirector();
        QueryBuilder builder = new SQLQueryBuilder();
        Query query = director.build(builder, "FROM Student", "WHERE code=1234");
        query.execute();
    }
}

```

Director quyết định cách dùng các thành phần xây dựng nên đối tượng Product, trong lúc loại ConcreteBuilder được lựa chọn mới thực sự tạo ra các thành phần này. Bạn có thể tạo thêm nhiều loại ConcreteBuilder khác, ví dụ NoSQLQueryBuilder.

2. Liên quan

- Abstract Factory: Builder tập trung vào các bước xây dựng đối tượng phức tạp, trong khi Abstract Factory tập trung vào việc tạo một họ các đối tượng chỉ trong một bước, giữ vai trò trung tâm khi khởi tạo đối tượng. Director có thể dùng Abstract Factory để tạo các đối tượng Builder.
- Template Method: Builder thường được cài đặt bằng cách dùng Template Method.
- Composite: Builder thường được dùng để xây dựng các đối tượng Composite.

3. Java API

Như một áp dụng của mẫu thiết kế Builder, đoạn code sau dùng SAXParser (Director) của JAXP để phân tích tập tin XML với lớp xử lý MySAXHandler do ta viết (ConcreteBuilder) dẫn xuất từ org.xml.sax.helpers.DefaultHandler (Builder).

```

SAXParser parser = factory.newSAXParser(); // Director
MySAXHandler handler = new MySAXHandler(); // ConcreteBuilder
parser.parse("books.xml", handler); // thiết lập Builder cho Director và xử lý
ArrayList<Book> books = handler.getBooks(); // lấy Product được tạo

```

Lớp java.lang.StringBuilder và lớp java.lang.StringBuffer cũng là các Builder, có phương thức tạo (append) trả về chính thể hiện của nó, giúp xây dựng đối tượng bằng cách khởi tạo từng phần.

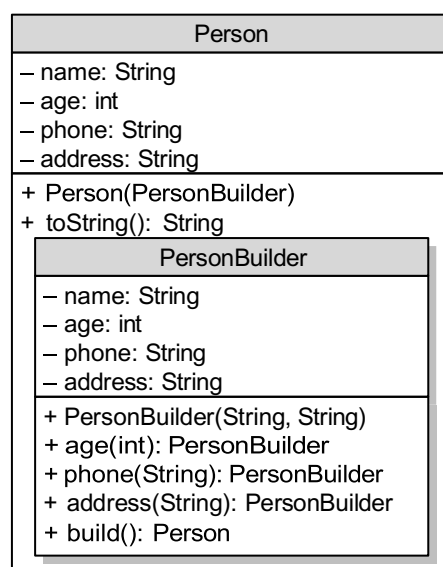
4. Sử dụng

Ta muốn:

- Quá trình xây dựng nên đối tượng độc lập với các thành phần tạo nên đối tượng.
- Dễ dàng thêm các cài đặt mới, tức các Builder mới.
- Kiểm soát linh hoạt hơn quá trình xây dựng đối tượng: vấn đề nhiều constructor và thứ tự tạo thành các phần của đối tượng.
- Giải quyết vấn đề constructor có quá nhiều tham số.

5. Bài tập

Thông tin đăng ký của một cá nhân có phần bắt buộc (name) và phần tùy chọn (age, phone, address). Phần tùy chọn có thể được bổ sung sau. Để tránh viết nhiều constructor với số tham số tăng dần (telescoping constructors) và có thể xây dựng đối tượng từng phần, áp dụng mẫu thiết kế Builder để thực hiện yêu cầu trên.



Factory Method

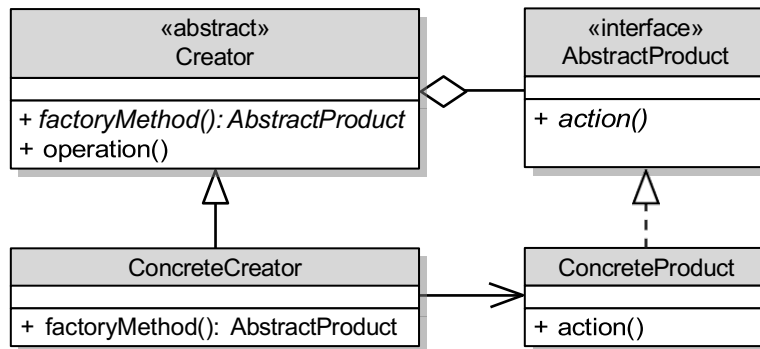
Delegate object creation 

Mẫu thiết kế Factory Method định nghĩa một giao diện trừu tượng dùng tạo một đối tượng, nhưng nó ủy nhiệm cho lớp dẫn xuất từ nó quyết định đối tượng thuộc lớp cụ thể nào sẽ được tạo. Do đặc điểm này, Factory Method còn gọi là Virtual Constructor.

Đối tượng tạo Creator, là một đối tượng trừu tượng *chứa phương thức (factory method) dùng tạo ra một đối tượng trừu tượng khác*, gọi là AbstractProduct. AbstractProduct có nhiều kiểu dẫn xuất khác nhau (ConcreteProduct). Vì Creator không biết kiểu ConcreteProduct cụ thể nào được tạo ra nên nó trì hoãn nhiệm vụ tạo đối tượng cho lớp con của nó thực hiện.

Trì hoãn tạo đối tượng có ý nghĩa, vì khi dùng toán tử new để tạo đối tượng, bạn phải *xác định ngay kiểu của đối tượng* được tạo. Khi dùng mẫu thiết kế Factory Method, bạn tạo đối tượng nhưng ủy nhiệm cho lớp khác quyết định kiểu của đối tượng đó. Trong OOP, thừa kế và viết lại phương thức là để thay đổi hành vi của lớp. Với mẫu thiết kế Factory Method, hành vi được thay đổi là tạo đối tượng, thừa kế Creator và viết lại phương thức factory để thay đổi cách tạo đối tượng.

1. Cài đặt



- Creator: lớp trừu tượng chứa phương thức trừu tượng factoryMethod() dùng tạo ra AbstractProduct. Phương thức này cũng có thể được tham số hóa để tạo ra nhiều kiểu AbstractProduct hoặc cài đặt sẵn để tạo ra đối tượng trừu tượng mặc định.

Creator cũng có thể chứa phương thức operation() có sử dụng AbstractProduct được tạo ra.

- ConcreteCreator: lớp con của Creator, cài đặt cụ thể phương thức factoryMethod() để sinh ra loại ConcreteProduct được chọn.

- AbstractProduct: giao diện của đối tượng được tạo ra từ phương thức factoryMethod(), có nhiều kiểu dẫn xuất.

- ConcreteProduct: cài đặt giao diện AbstractProduct, là đối tượng cụ thể được tạo ra từ phương thức factoryMethod() của ConcreteCreator.

```

abstract class Creator {
    public abstract SortAlgorithm createAlgorithms(int type);

    public void sortArray(int type, Comparable... array) {
        createAlgorithms(type).sort(array);
    }
}
  
```

```

// (2) ConcreteCreator
class SortAlgorithmCreator extends Creator {
    public static final int INSERTION_SORT = 0;
    public static final int SELECTION_SORT = 1;
    @Override public SortAlgorithm createAlgorithms(int type) {
        switch (type) {
            case INSERTION_SORT: return new InsertionSort();
            case SELECTION_SORT: return new SelectionSort();
            default: return null;
        }
    }
}
  
```

```

// (3) AbstractProduct
interface SortAlgorithm<T extends Comparable<? super T>> {
    void sort(T... array);
}
  
```

```

// (4) ConcreteProduct
class InsertionSort<T extends Comparable<? super T>> implements SortAlgorithm<T> {
    @Override public void sort(Comparable... array) {
        Comparable temp;
        int i, j;
        for (i = 1; i < array.length; i++) {
            temp = array[i];
            for (j = i; j > 0 && temp.compareTo(array[j - 1]) < 0; j--) array[j] = array[j - 1];
            array[j] = temp;
        }
    }
}
  
```

```

    }
}

class SelectionSort<T extends Comparable<? super T>> implements SortAlgorithm<T> {
    @Override public void sort(Comparable... array) {
        int i, j, least;
        for (i = 0; i < array.length - 1; ++i) {
            for (j = i + 1, least = i; j < array.length; ++j)
                if (array[j].compareTo(array[least]) < 0) least = j;
            if (least != i) {
                Comparable temp = array[least]; array[least] = array[i]; array[i] = temp;
            }
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Factory Method ---");
        int[] a = { 1, 8, 3, 6, 5, 4, 7, 2, 9 };
        Integer[] o = new Integer[a.length];
        for (int i = 0; i < o.length; i++) o[i] = a[i];
        Creator creator = new SortAlgorithmCreator();
        creator.sortArray(SortAlgorithmCreator.SELECTION_SORT, o);
        for (Integer i : o) System.out.println(i);
    }
}

```

Client gọi phương thức sortArray() được tham số hóa để chỉ định thuật toán sắp xếp. Phương thức này gọi phương thức factory đa hình createAlgorithm() của Creator để tạo đối tượng SortAlgorithm yêu cầu. Lớp con của Creator, SortAlgorithmCreator, mới là lớp thật sự tạo ra đối tượng SortAlgorithm cụ thể: InsertionSort hoặc SelectionSort.

2. Liên quan

- Template Method: Factory Method đôi khi được gọi bởi Template Method.
- Prototype: các Prototype không cần dẫn xuất Creator, chúng cần một tác vụ khởi tạo trong lớp Product, Creator dùng tác vụ khởi tạo này để khởi tạo đối tượng.
- Abstract Factory: thường dễ nhầm lẫn giữa Abstract Factory và Factory Method. Khác biệt chủ yếu giữa chúng:
 - + Factory Method: quan tâm đến *phương thức factory*, tức phương thức sinh đối tượng. Ý tưởng chính là khai báo phương thức factory ở giao diện trừu tượng và cài đặt nó ở lớp dẫn xuất; để lớp dẫn xuất quyết định lớp đối tượng cụ thể được tạo từ phương thức factory.
 - + Abstract Factory: quan tâm đến *đối tượng factory*. Ý tưởng chính là tạo một lớp Factory đóng gói nhiều phương thức tạo (creational method, phân biệt với phương thức factory); Client ủy nhiệm cho phương thức tạo cụ thể tạo đối tượng. Các phương thức tạo trong Abstract Factory thường được cài đặt với Factory Method.
- Iterator: thường được tạo từ một Factory Method.

3. Java API

Giao diện Collection của Java có khai báo một phương thức tạo ra một Iterator, dùng để duyệt các phần tử của nó:

```
Iterator iterator();
```

Mỗi lớp dẫn xuất từ Collection cài đặt phương thức iterator() theo cách khác nhau, do đối tượng Iterator của chúng có cơ chế hoạt động hoàn toàn khác nhau. Khả năng đa hình cho phép lời gọi: Iterator iterator = collection.iterator(); gọi phương thức iterator() của lớp dẫn xuất cụ thể để tạo ra Iterator cụ thể phù hợp với loại collection dẫn xuất đó. Điều này linh động hơn việc tạo Iterator từ constructor.

Trong trường hợp này: Creator là Collection, ConcreteCreator là các lớp dẫn xuất của Collection (LinkedList, ArrayList, Queue), AbstractProduct là Iterator, ConcreteProduct là lớp dẫn xuất từ Iterator (thường vô danh), phương thức tạo factoryMethod() là phương thức iterator().

Lớp java.util.Calendar cũng là một Creator, phương thức tạo của nó là getInstance().

4. Sử dụng

Ta có:

- Biết rõ khi nào đối tượng được tạo nhưng chưa biết rõ thông tin về đối tượng được tạo. Ví dụ kiểu đối tượng được tạo phụ thuộc vào tham số được nhập vào.
- Việc tạo đối tượng trở nên phức tạp. Ví dụ việc khởi tạo đối tượng phụ thuộc vào nhiều đối tượng khác, phải xây dựng thêm các đối tượng liên quan.

Ta muốn:

- Linh hoạt trong việc tạo đối tượng. Hệ thống các lớp được tạo có thể thay đổi tự động.

Xem xét sử dụng:

- Abstract Factory, Prototype, hoặc Builder. Chúng phức tạp hơn, nhưng linh hoạt hơn.
- Prototype, lưu tập các đối tượng được nhân bản từ Abstract Factory.

5. Bài tập

a) Chương trình vẽ dùng lớp Creator để sinh và quản lý các hình vẽ (Shape). Có hai loại hình vẽ: Rectangle (với thông tin là chiều dài w và chiều rộng h) và Circle (với thông tin là bán kính r). Chúng dùng phương thức draw() để vẽ.

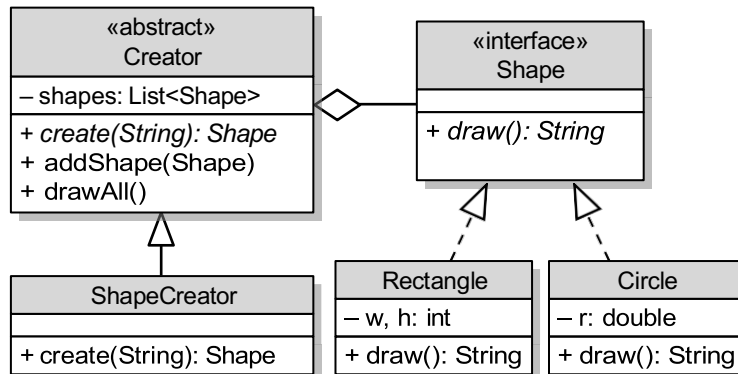
Phương thức create() của lớp Creator nhận chuỗi chứa thông tin hình vẽ, có định dạng:

#rectan w, h. Ví dụ #rectan 8, 5

#circle r. Ví dụ #circle 3.7

Với thông tin này, nó tạo và lưu trữ loại Shape tương ứng.

Lớp Creator cũng có phương thức drawAll() dùng vẽ tất cả hình nó lưu trữ.



b) Cho lớp PaymentService, constructor của nó tạo một thể hiện của lớp FinancialTrustCCP, là một dịch vụ xử lý thẻ (Credit Card Processing) do bên thứ ba cung cấp.

```

public class PaymentService {
    private final static String recipientID = "123-456-789";
    private FinancialTrustCCP ccp; // dịch vụ xử lý thẻ của bên thứ ba

    public PaymentService() {
        ccp = new FinancialTrustCCP();
    }

    public void pay(String senderID, String amount) { // chuyển tiền từ sender đến recipient
        boolean approved = ccp.post(senderID, recipientID, amount);
        if (approved) // ...
        else // ...
    }
}
  
```

Đối tượng được tạo FinancialTrustCCP liên kết quá chặt với đối tượng dùng nó là PaymentService nên vi phạm nguyên tắc OCP. Không thể nào mở rộng giải pháp hiện tại, ví dụ làm việc với một dịch vụ xử lý thẻ khác, hoặc kiểm thử với mock object, mà không phải thay đổi code.

Bạn hãy giải quyết vấn đề này bằng cách áp dụng mẫu thiết kế Factory Method. Ý tưởng là thay thế việc tạo trực tiếp đối tượng FinancialTrustCCP bằng một Factory Method như sau: CCPService createCCPService();

CCPService do phương thức trả về là interface mà FinancialTrustCCP, các dịch vụ xử lý thẻ khác hoặc mock object, phải cài đặt.

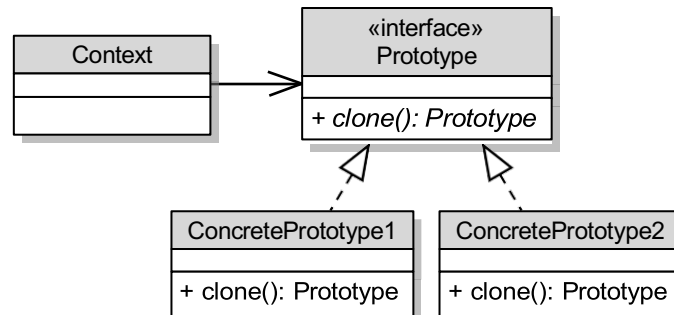
Prototype

Create object on prototype 

Mẫu thiết kế Prototype dùng một thể hiện nguyên mẫu (prototypical instance) để chỉ định loại đối tượng sẽ được tạo. Nói cách khác, mẫu thiết kế Prototype tạo các đối tượng mới bằng cách *nhân bản một thể hiện nguyên mẫu là chính nó*.

Ta không tạo ra các đối tượng trực tiếp từ lớp mà sao chép chúng từ đối tượng nguyên mẫu đang tồn tại và thay đổi thuộc tính của chúng nếu cần. Mục đích là thu giảm chi phí khi khởi tạo (ví dụ truy xuất cơ sở dữ liệu) cho các đối tượng có cùng kiểu.

1. Cài đặt



- Prototype: khai báo giao diện để nhân bản chính nó. Mỗi Prototype là một đối tượng factory, có thể tạo đối tượng giống chính nó, code dùng tạo đối tượng đặt trong phương thức clone() của nó.

- ConcretePrototype: cài đặt cụ thể tác vụ clone() để nhân bản với dữ liệu lựa chọn.

Đối tượng nhân bản (cloned object) là một đối tượng mới, được tạo ra bằng cách sao chép sâu (deep copy) nên nó tách rời khỏi đối tượng gốc ban đầu. Sau đó, tùy biến đối tượng được nhân bản.

```
import java.util.*;
```

// (1) Prototype

```
abstract class ColorPrototype implements Cloneable {
    protected List<Integer> list = new ArrayList<>();
    protected ColorPrototype(Integer... colors) {
        list.addAll(Arrays.asList(colors));
    }

    @Override public String toString() {
        return list.toString();
    }
}
```

// (2) ConcretePrototype

```
class Color extends ColorPrototype {
    public Color(Integer... colors) {
        super(colors);
    }

    public Color setColor(Integer... colors) {
        for (int i = 0; i < colors.length; i++)
            list.set(i, colors[i]);
        return this;
    }

    @Override public Color clone() throws CloneNotSupportedException {
        System.out.println("[LOG]: Cloning");
        Color clone = (Color) (ColorPrototype) super.clone();
        // deep copy
        clone.list = new ArrayList<>(list);
        return clone;
    }
}
```

// (3) Context

```
class ColorPalette {
    private HashMap<String, ColorPrototype> palette = new HashMap<>();
    public HashMap<String, ColorPrototype> getPalette() {
        return palette;
    }

    public ColorPalette addColor(String name, ColorPrototype color) {
        palette.put(name, color);
        return this;
    }
}
```



```

public void showPalette() {
    for (String key : palette.keySet()) {
        System.out.printf("%s: %s%n", palette.get(key), key);
    }
}

public class Client {
    public static void main(String[] args) throws CloneNotSupportedException {
        System.out.println("--- Prototype Pattern ---");
        Color base = new Color(0, 0, 0);
        Color red = base.clone().setColor(255, 0, 0);
        Color green = base.clone().setColor(0, 255, 0);
        Color blue = base.clone().setColor(0, 0, 255);
        new ColorPalette()
            .addColor("red", red)
            .addColor("green", green)
            .addColor("blue", blue)
            .showPalette();
    }
}

```

ColorPalette lưu trữ ba đối tượng Color: "red", "green" và "blue"; chúng được "nhân bản" từ Color gốc (base) rồi thiết đặt lại. Chỉ có thực thể gốc "base" này được tạo từ constructor, dùng toán tử new.

2. Liên quan

- Abstract Factory: Prototype chính là một đối tượng factory. Vì vậy có thể lưu một họ Prototype để nhân bản và trả về đối tượng được tạo.
- Composite và Decorator: tạo Prototype rồi "phức hợp" thêm bằng Composite hoặc "trang trí" thêm bằng Decorator.

3. Java API

Phương thức clone() của java.lang.Object là phương thức tạo, trả về một thể hiện khác của chính đối tượng, có cùng thuộc tính của đối tượng gọi. Lưu ý phương thức clone() chỉ sao chép bề mặt (shallow copy).

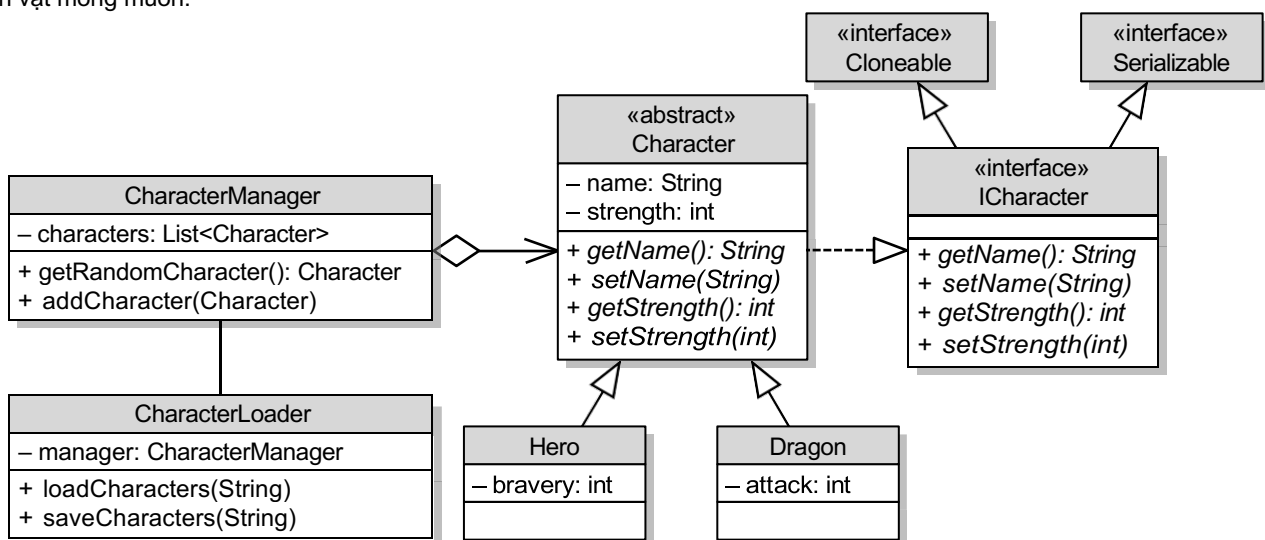
4. Sử dụng

Ta muốn:

- Che giấu quá trình tạo các lớp cụ thể với Client.
- Dùng Prototype để thêm và loại các lớp mới trong thời gian chạy.
- Giữ số lượng các lớp trong hệ thống ở mức tối thiểu.
- Thích ứng với thay đổi cấu trúc dữ liệu trong thời gian chạy.

5. Bài tập

Trong một trò chơi, mẫu (prototype) của các nhân vật (Character) sẽ xuất hiện trong trò chơi được lưu trong tập tin, CharacterLoader sẽ đọc chúng (bằng ObjectOutputStream) và lưu vào danh sách characters của CharacterManager bằng phương thức addCharacter() của lớp này. Khi muốn tạo thêm nhân vật, thay vì phải đọc tập tin, CharacterManager gọi phương thức getRandomCharacter(), phương thức này lấy một prototype ngẫu nhiên trong danh sách characters để nhân bản (clone) ra nhân vật mong muốn.



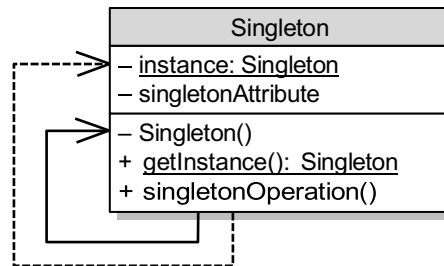
Singleton

Single object instances 

Mẫu thiết kế Singleton đảm bảo rằng một lớp chỉ có *một thể hiện* (instance) duy nhất. Do thể hiện này có tiềm năng sử dụng trong suốt chương trình, nên mẫu thiết kế Singleton cũng cung cấp *một điểm truy cập toàn cục* đến nó.

Hộp thoại Find là một ví dụ điển hình cho mẫu thiết kế Singleton. Dù bạn chọn menu hoặc nhấn Ctrl+F nhiều lần, tại bất kỳ nơi nào trong ứng dụng, chỉ một hộp thoại duy nhất xuất hiện.

1. Cài đặt



Mẫu thiết kế Singleton đơn giản và dễ áp dụng, chỉ cần bổ sung vài dòng lệnh trong lớp muốn chuyển thành Singleton.

- Dữ liệu thành viên instance (private và static) là đối tượng duy nhất của lớp Singleton.
- Constructor của lớp Singleton được định nghĩa thành protected hoặc private để ngăn người dùng tạo thực thể trực tiếp từ bên ngoài lớp.

- Phương thức getInstance() dùng khởi tạo đối tượng duy nhất, định nghĩa thành public và static. Client chỉ dùng getInstance() để tạo đối tượng cho lớp Singleton.

- Thực hiện khởi tạo chậm (lazy initialization) trong getInstance(): chỉ khi gọi phương thức getInstance() mới khởi tạo đối tượng. Phương thức này trả về một thể hiện mới hay một null tùy thuộc vào một biến boolean dùng như cờ hiệu báo xem lớp Singleton đã tạo thể hiện hay chưa.

Trong chế độ đa luồng (multithreading), mẫu thiết kế Singleton có thể làm việc không tốt: do getInstance() không an toàn thread, hai thread có thể gọi phương thức sinh đối tượng cùng một thời điểm và hơn một thể hiện sẽ được tạo ra. Giải quyết bằng đồng bộ (synchronized) phương thức getInstance() để an toàn thread sẽ dẫn đến giảm hiệu suất chương trình.

Có nhiều giải pháp cho vấn đề này:

- double-checked locking

Kiểm tra sự tồn tại thể hiện của lớp, với sự hỗ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo volatile cho instance để tránh lỗi làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

```
// (1) Singleton
class Singleton {
    private static Singleton instance;

    private Singleton() { }

    private synchronized static void createInstance() {
        if (instance == null) instance = new Singleton();
    }

    public static Singleton getInstance() {
        if (instance == null) createInstance();
        return instance;
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Singleton Pattern ---");
        Singleton single1 = Singleton.getInstance();
        Singleton single2 = Singleton.getInstance();
        if (single1.equals(single2)) {
            System.out.println("Unique Instance");
        }
    }
}
```

Một cách cài đặt khác:

```
class Singleton {
    private static volatile Singleton instance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) instance = new Singleton();
            }
        }
        return instance;
    }
}
```

```

    }
}
return instance;
}
}

```

- eager initialization

Thực thể Singleton là biến static và final, được khởi tạo sớm khi lớp lần đầu được nạp vào JVM.

// (1) Singleton

```

class Singleton {
    // eager initialization
    private static final Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton getInstance() {
        return instance;
    }
}

```

```

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Singleton Pattern ---");
        Singleton single1 = Singleton.getInstance();
        Singleton single2 = Singleton.getInstance();
        if (single1.equals(single2)) {
            System.out.println("Unique Instance");
        }
    }
}

```

- class loader (static block initialization)

// (1) Singleton

```

class Singleton {
    private static class SingletonHelper {
        static final Singleton instance = new Singleton();
    }

    private Singleton() { }

    public static Singleton getInstance() {
        return SingletonHelper.instance;
    }
}

```

```

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Singleton Pattern ---");
        Singleton single1 = Singleton.getInstance();
        Singleton single2 = Singleton.getInstance();
        if (single1.equals(single2)) {
            System.out.println("Unique Instance");
        }
    }
}

```

- enum singleton

// (1) Singleton

```

enum Singleton {
    SINGLETON;
}

```

```

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Singleton Pattern ---");
        Singleton single1 = Singleton.SINGLETON;
        Singleton single2 = Singleton.SINGLETON;
        if (single1.equals(single2)) {
            System.out.println("Unique Instance");
        }
    }
}

```

2. Liên quan

- Abstract Factory: thường là Singleton để trả về các đối tượng factory duy nhất.
- Builder: dùng xây dựng một đối tượng phức tạp, trong đó Singleton được dùng để tạo một đối tượng truy cập tổng quát (Director).
- Prototype: dùng để sao chép một đối tượng, hoặc tạo ra một đối tượng khác từ Prototype của nó, trong đó Singleton được dùng để chắc chắn chỉ có một Prototype.

3. Java API

- Lớp `java.lang.Runtime` là lớp Singleton, để lấy được đối tượng duy nhất của nó, ta gọi phương thức `getRuntime()`.
 - Lớp `java.awt.Desktop` cũng là lớp Singleton, tạo đối tượng duy nhất bằng phương thức `getDesktop()`.
 - Lớp `java.util.logging.Logger`, với phương thức `getLogger()`.
- Tuy nhiên, lớp Singleton không phổ biến như ta nghĩ, không nên lạm dụng nó mà chỉ áp dụng với lớp cần bảo đảm có duy nhất một thể hiện. Lớp `java.lang.Math` và lớp `java.util.Calendar` chẳng hạn, không phải là lớp Singleton.

4. Sử dụng

Ta muốn:

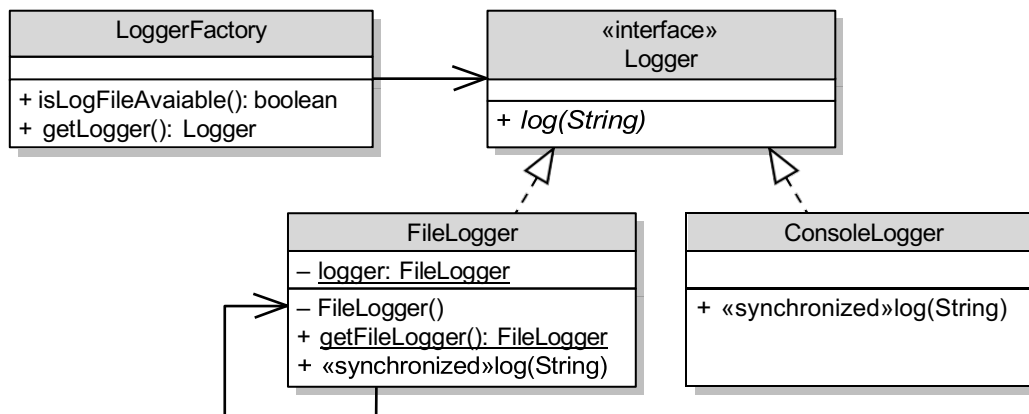
- Đảm bảo rằng chỉ có một thể hiện của lớp.
- Quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
- Quản lý số lượng thể hiện của một lớp. Trường hợp này không nhất thiết chỉ có một thể hiện, bạn cần kiểm soát số lượng thể hiện trong giới hạn chỉ định.

5. Bài tập

Để xây dựng hệ thống ghi nhận thông tin, người ta dùng giao diện `Logger` với phương thức `log()`. Có hai loại `Logger`: `ConsoleLogger` xuất thông tin ghi nhận ra màn hình, `FileLogger` lưu thông tin ghi nhận vào tập tin `log.txt`, chèn dòng thông tin mới vào phía đầu tập tin.

`LoggerFactory` chịu trách nhiệm quyết định loại `Logger` sẽ sử dụng. Nó xem xét mục `FileLogging` trong tập tin `logger.properties`, nếu mục này là `ON`, `FileLogger` sẽ được tạo và sử dụng; ngược lại, `ConsoleLogger` sẽ được sử dụng.

Vì chỉ có một tập tin nhật ký vật lý được tham chiếu bởi `FileLogger` nên `FileLogger` phải là một Singleton. Viết các lớp cần thiết và áp dụng mẫu thiết kế Singleton để chuyển `FileLogger` thành một Singleton.



Structural

Adapter

Adapt from one to another 

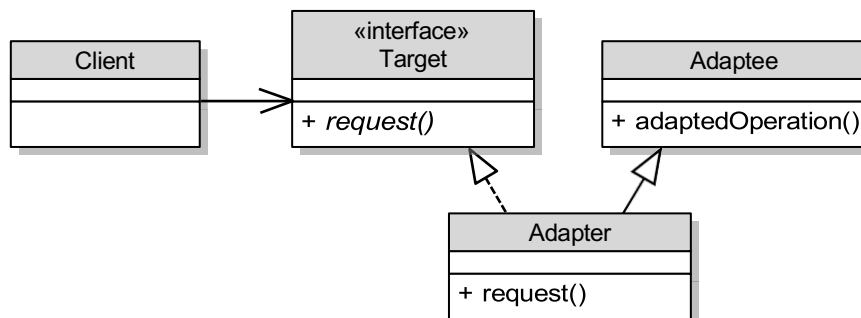
Mẫu thiết kế Adapter giữ vai trò trung gian giữa hai lớp, *chuyển đổi giao diện* của một hay nhiều lớp có sẵn thành một giao diện khác, tương thích với lớp đang viết. Adapter giải quyết vấn đề không tương thích của hai giao diện, điều này cho phép các lớp có các giao diện khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua *giao diện chuyển tiếp trung gian*, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.

Mẫu thiết kế Adapter còn gọi là Wrapper do cung cấp một giao diện "bọc ngoài" tương thích cho một hệ thống có sẵn, hệ thống có sẵn này có dữ liệu và hành vi phù hợp nhưng có giao diện không tương thích với lớp đang viết.

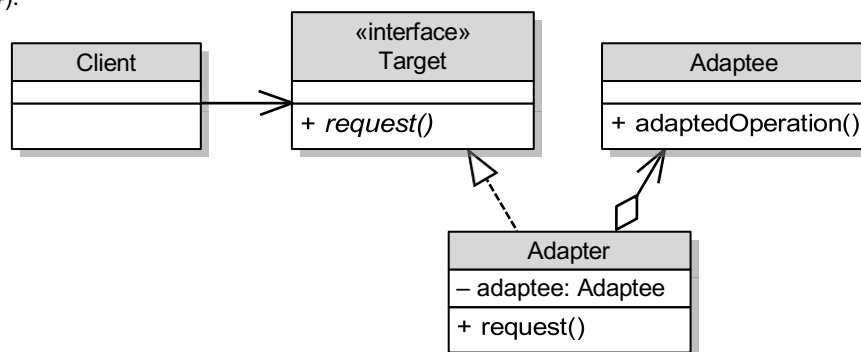
1. Cài đặt

Có hai cách để cài đặt mẫu thiết kế Adapter:

- Tiếp hợp lớp (dùng thừa kế – inheritance): một lớp mới (Adapter) sẽ *kế thừa* lớp có sẵn với giao diện không tương thích (Adaptee), đồng thời *cài đặt giao diện* mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của giao diện người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có giao diện không tương thích. Tiếp hợp lớp đơn giản nhưng dễ gặp trường hợp đụng độ tên phương thức.



- Tiếp hợp đối tượng (dùng tích hợp – composition): một lớp mới (Adapter) sẽ *tham chiếu* đến một (hoặc nhiều) đối tượng của lớp có sẵn với giao diện không tương thích (Adaptee), đồng thời *cài đặt giao diện* mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của giao diện người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng¹ thuộc lớp có giao diện không tương thích. Tiếp hợp đối tượng tránh được vấn đề đa thừa kế, không có trong các ngôn ngữ hiện đại (Java, C#).



Các thành phần tham gia:

- Target: định nghĩa giao diện Client đang làm việc (phương thức request()).
- Adaptee: có giao diện không tương thích (phương thức adaptedOperation()), cần được tiếp hợp để sử dụng được.
- Adapter: lớp tiếp hợp, giúp giao diện đang làm việc tiếp hợp được với giao diện không tương thích. Phương thức request() của nó ủy nhiệm việc thực hiện cho Adaptee bằng cách gọi adaptedOperation() của Adaptee. Adapter có thể chứa mã bổ sung để phù hợp với nhu cầu của Client. Có thể cung cấp nhiều Adapter.

```
// (1) Target
interface Target {
    double getArea(Point topleft, Point rightbottom);
}
```

```
class Point {
    double x, y;
    public Point (double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

```
// (2) Adaptee
class Adaptee {
```

¹ Nếu Adapter tham chiếu đến nhiều đối tượng Adaptee, có thể thực hiện được tính đa hình (polymorphism) trong Adapter.

```

public double getArea(double x1, double y1, double x2, double y2) {
    return Math.abs((x2 - x1) * (y2 - y1));
}
}

// (3) Adapter
class Adapter extends Adaptee implements Target {
    @Override public double getArea(Point topleft, Point rightbottom) {
        return getArea(topleft.x, topleft.y, rightbottom.x, rightbottom.y);
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Adapter Pattern ---");
        Target target = new Adapter();
        System.out.printf("Area = %.1f\n", target.getArea(new Point(1, 4), new Point(5, 1)));
    }
}

```

Client định gọi phương thức `getArea()` của `Adaptee` để tính diện tích hình chữ nhật nhưng không thực hiện được do khác danh sách tham số. Adapter giải quyết vấn đề này: phương thức `getArea()` của Adapter có signature phù hợp với lời gọi của Client, và khi thực thi nó chuyển tiếp lời gọi đến phương thức `getArea()` của `Adaptee`. Adapter chỉ đóng vai trò chuyển đổi giao diện, nó ủy nhiệm cho `Adaptee` thực hiện lời gọi.

2. Liên quan

- Bridge: có cấu trúc tương tự nhưng mục tiêu khác (tách một giao diện khỏi phần cài đặt).
- Decorator: bổ sung thêm chức năng nhưng không làm thay đổi giao diện, trong mẫu thiết kế Decorator, một Adapter sẽ phối hợp hai đối tượng khác nhau.
- Proxy: định nghĩa một giao diện đại diện cho các đối tượng khác mà không làm thay đổi giao diện của các đối tượng được đại diện, điều này thực hiện được nhờ các Adapter.

3. Java API

Mẫu thiết kế Adapter dùng phổ biến trong Java AWT (`WindowAdapter`, `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter`, `KeyAdapter`, `MouseAdapter` và `MouseMotionAdapter`), ...

Ví dụ: giao diện `WindowListener` có 7 phương thức. Khi lớp lắng nghe sự kiện (listener) của ta cài đặt giao diện này, cần phải cài đặt *tất cả* 7 phương thức xử lý sự kiện, dù một số phương thức chỉ cài đặt rỗng do không dùng đến loại sự kiện đó. Lớp `WindowAdapter` cài đặt giao diện `WindowListener`, cài đặt sẵn và rỗng cả 7 phương thức. Như vậy nếu lớp lắng nghe sự kiện của ta thừa kế lớp `WindowAdapter`, chỉ viết lại (override) chỉ những phương thức ta muốn thay đổi.

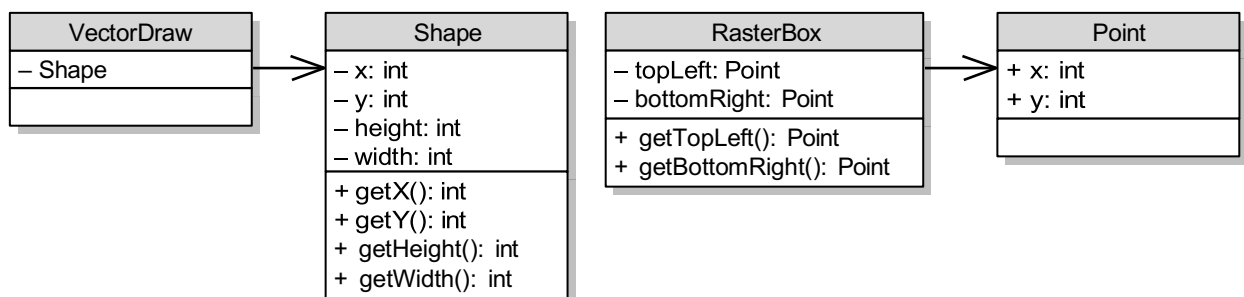
4. Sử dụng

Ta muốn:

- Sử dụng một lớp đã tồn tại trước đó nhưng giao diện sử dụng không phù hợp như mong muốn, ta lại không có mã nguồn để sửa đổi giao diện đó.
- Sử dụng một lớp, nhưng lớp này được tạo ra với mục đích chung, nên không thích hợp cho việc tạo một giao diện đặc thù.
- Có sự chuyển đổi giao diện từ nhiều nguồn khác nhau.
- Tiếp hợp nhiều đối tượng cùng một lúc, nhưng giao diện mong muốn không phải là interface mà là một lớp trừu tượng.

5. Bài tập

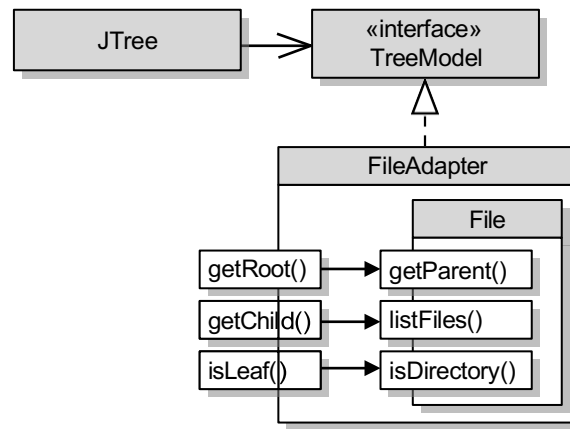
a) Lớp `VectorDraw` của khách hàng đã sử dụng lớp `Shape`. Áp dụng mẫu thiết kế Adapter để cho lớp `VectorDraw` sử dụng được thêm lớp `RasterBox` (và lớp `Point` mà `RasterBox` sử dụng).



b) Duyệt hệ thống tập tin và hiển thị lên JTree theo sơ đồ sau:

JTree (View) \leftrightarrow TreeModel (Model) \leftrightarrow **FileAdapter** \leftrightarrow File

Bạn hãy sử dụng mẫu thiết kế Adapter, chuyển đổi giao diện File thành giao diện TreeModel.



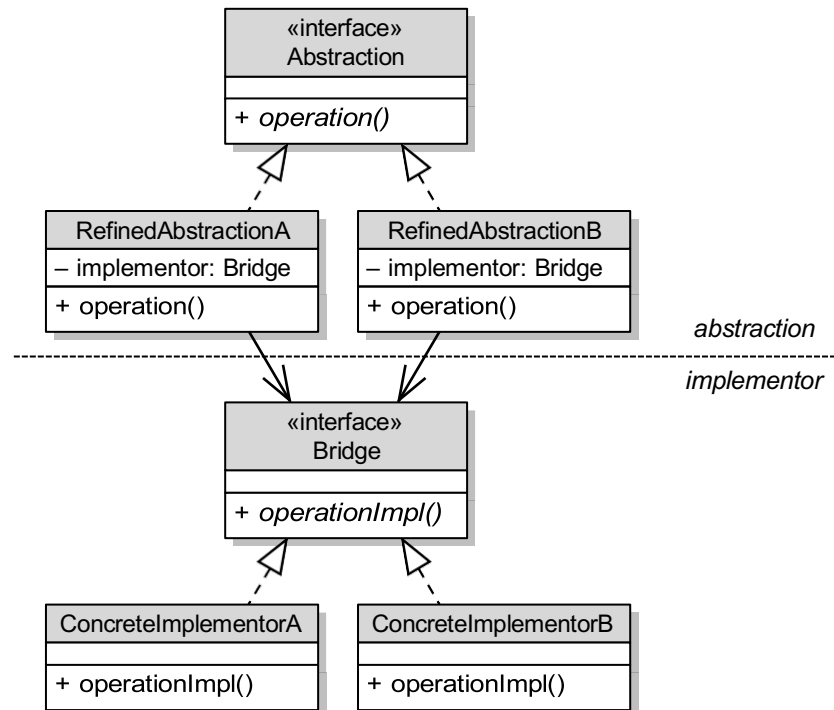
Bridge

Decouple abstraction from implementation 

Mẫu thiết kế Bridge dùng tách một đối tượng phức tạp thành hai thành phần: phần trừu tượng (Abstraction) và phần cài đặt (Implementor) thường nặng tính kỹ thuật, giúp bạn có thể thay đổi hai thành phần này, gọi là "hai phía của cầu", một cách độc lập. Mẫu thiết kế Bridge *cung cấp một cầu nối (bridge) giữa phần trừu tượng và phần cài đặt*. Do đặc điểm này, Bridge còn gọi là Handle/Body.

Mẫu thiết kế Bridge cần khi một phiên bản mới của ứng dụng được dùng để thay thế phiên bản cũ, nhưng phiên bản cũ vẫn còn chạy trên cơ sở Client cũ. Code của Client sẽ không thay đổi do tương thích với trừu tượng cũ.

1. Cài đặt



- Abstraction: giao diện trừu tượng mà Client nhìn thấy.
- RefinedAbstraction: dẫn xuất của Abstraction để cải tiến phần trừu tượng (không có nghĩa Concrete Abstraction).
- Bridge: còn gọi là Implementor, giao diện cầu nối giữa phần trừu tượng và phần cài đặt.
- ConcreteImplementor: cài đặt cụ thể cho phần trừu tượng Abstraction, đã được Bridge tách ra để tùy biến riêng. Chú ý giao diện của nó không nhất thiết phải giống với Abstraction, Bridge sẽ giải quyết sự khác biệt này.

// (1) Abstraction

```

abstract class Shape {
    protected Drawing implementor;
    abstract public void draw();

    protected Shape(Drawing dp) {
        this.implementor = dp;
    }
}
  
```

// (2) RefinedAbstraction

```

class Rectangle extends Shape {
    private double x1, y1, x2, y2;
    public Rectangle(Drawing dp, double x1, double y1, double x2, double y2) {
        super(dp);
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }

    @Override public void draw() {
        implementor.drawLine(x1, y1, x2, y1);
        implementor.drawLine(x2, y1, x2, y2);
        implementor.drawLine(x2, y2, x1, y2);
        implementor.drawLine(x1, y2, x1, y1);
    }
}
  
```

```

class Circle extends Shape {
  
```

```

private double x, y, r;
public Circle(Drawing dp, double x, double y, double r) {
    super(dp);
    this.x = x;
    this.y = y;
    this.r = r;
}

@Override public void draw() {
    implementor.drawCircle(x, y, r);
}
}

// (3) Bridge
abstract class Drawing {
    abstract public void drawLine(double x1, double y1, double x2, double y2);
    abstract public void drawCircle(double x, double y, double r);
}

// (4) ConcreteImplementor
class V1Drawing extends Drawing {
    @Override public void drawLine(double x1, double y1, double x2, double y2) {
        System.out.printf("(%.1f, %.1f)-----(%.1f, %.1f)%n", x1, y1, x2, y2);
    }

    @Override public void drawCircle(double x, double y, double r) {
        System.out.printf("(%.1f, %.1f)----[%.1f]--->)%n", x, y, r);
    }
}

class V2Drawing extends Drawing {
    @Override public void drawLine(double x1, double y1, double x2, double y2) {
        System.out.printf("(%.1f, %.1f).....(%.1f, %.1f)%n", x1, y1, x2, y2);
    }

    @Override public void drawCircle(double x, double y, double r) {
        System.out.printf("(<----[%.1f]-----(%.1f, %.1f)%n", r, x, y);
    }
}

public class Client {
    public static void main(String argv[]) {
        System.out.println("--- Bridge Pattern ---");
        Drawing dp1 = new V1Drawing();
        Drawing dp2 = new V2Drawing();
        Shape[] shapes = { new Rectangle(dp1, 1, 1, 2, 2), new Circle(dp1, 2, 2, 3),
                           new Rectangle(dp2, 1, 1, 2, 2), new Circle(dp2, 2, 2, 3) };
        for (Shape shape : shapes) shape.draw();
    }
}

```

Client chỉ làm việc với phần trừu tượng, trong lúc cài đặt cụ thể của phần trừu tượng được thay đổi, phát triển độc lập. Bridge giữ vai trò cầu nối giữa phần trừu tượng và phần cài đặt cụ thể.

2. Liên quan

- Abstract Factory: Abstract Factory có thể tạo và cấu hình một Bridge cụ thể.
- Adapter: Adapter được áp dụng nếu hệ thống đã thiết kế xong, giúp cho các lớp không liên quan làm việc được với nhau. Trái ngược với Adapter, Bridge được thiết kế ngay từ đầu, cho phép phần trừu tượng và phần hiện thực được tùy biến độc lập, để có thể mở rộng hệ thống sau khi thiết kế.

3. Java API

Java AWT 1.1.x dùng mẫu thiết kế Bridge để tách biệt các component trừu tượng với phần cài đặt phụ thuộc hệ nền của chúng. Ví dụ, java.awt.Button hoàn toàn bằng Java trong lúc sun.awt.windows.WButtonPeer được cài đặt bằng code Windows gốc (native).

4. Sử dụng

Ta có:

- Phần cài đặt có khả năng thay đổi nhiều phiên bản, trong lúc không muốn thay đổi code của Client làm việc với phần trừu tượng.

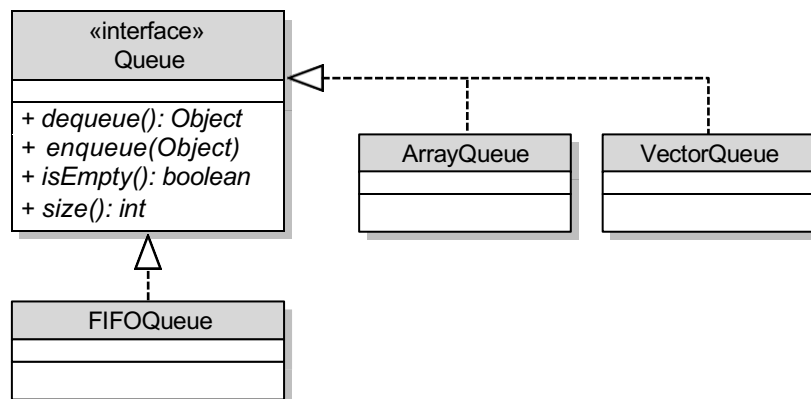
Ta muốn:

- Che giấu hoàn toàn phần cài đặt với Client.
- Tránh ràng buộc trực tiếp giữa phần trừu tượng và phần cài đặt.

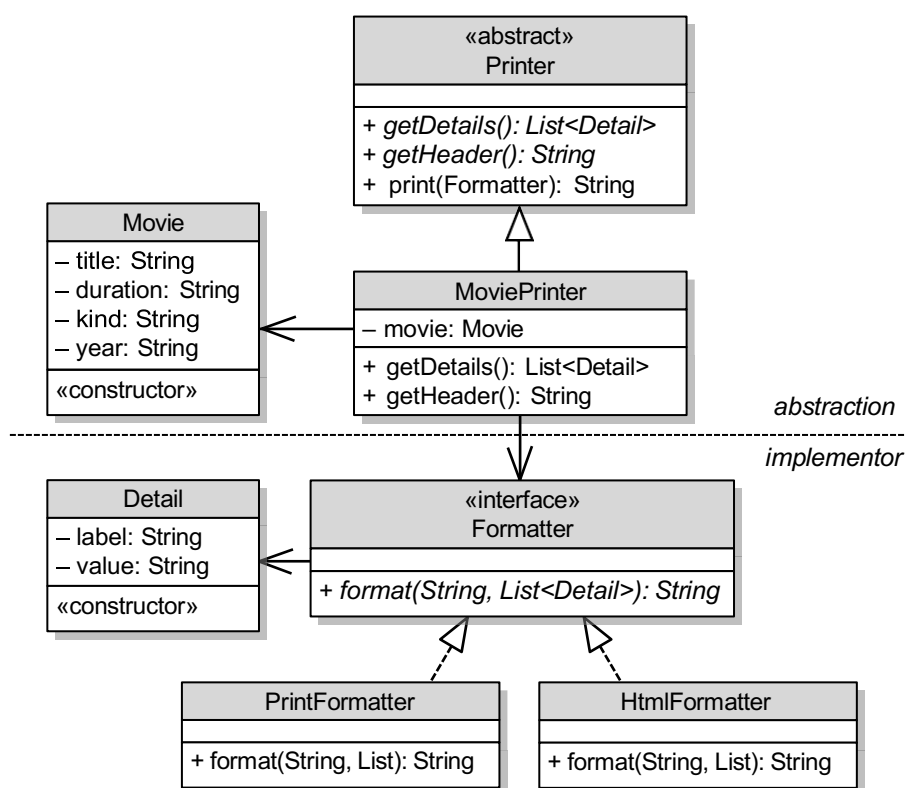
- Thay đổi phần cài đặt mà không cần biên dịch lại phần trừu tượng.
- Kết hợp các phần khác nhau của một hệ thống trong thời gian chạy.

5. Bài tập


a) Bạn cần mở rộng khái niệm Queue bằng cách thêm một phân lớp FIFOQueue. Hãy dùng mẫu thiết kế Bridge để thực hiện yêu cầu này.



b) Thực hiện các lớp được thiết kế theo mẫu thiết kế Bridge:



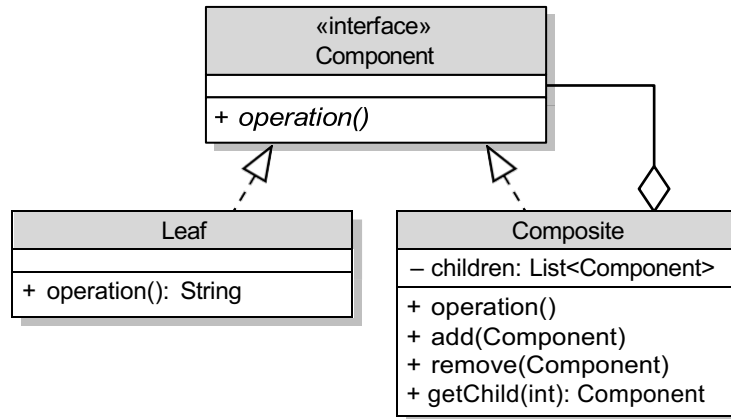
Composite

Uniformly treat tree objects 

Mẫu thiết kế Composite cung cấp một giao diện chung để xử lý với một đối tượng đơn theo cách tương tự như xử lý với một đối tượng phức hợp. Hai loại đối tượng (Component) này được *tổ chức như một cấu trúc cây*:

- Đối tượng đơn (primitive object) xem như node lá (Leaf, hoặc terminal) của cây.
 - Đối tượng phức hợp (Composite) xem như các node trong (non-terminal) của cây, có thể chứa các đối tượng đơn (node lá) hoặc các đối tượng phức hợp khác. Nói cách khác, đối tượng phức hợp chứa một nhóm các đối tượng Component.
- Ngoài tác vụ xử lý chung, đối tượng phức hợp còn có các tác vụ diễn hình để xử lý các đối tượng con của nó: thêm đối tượng (add), loại bỏ đối tượng (remove), lấy nhóm đối tượng con (getChild), hiển thị các đối tượng (print), tìm kiếm (find).

1. Cài đặt



- Component: giao diện khai báo tác vụ xử lý chung cho Leaf và Composite, là hành vi đa hình.
- Leaf: đối tượng đơn, không chứa các đối tượng con khác, tương tự như node lá của cây. Vì thế nó chỉ chứa tác vụ xử lý chung, không chứa các tác vụ dùng truy cập đối tượng con.
- Composite: đối tượng phức hợp, tương tự node trong của cây, chứa danh sách các đối tượng con (node lá hoặc node phức hợp). Ngoài tác vụ xử lý chung, còn có các phương thức cho phép truy cập các đối tượng con.

```
import java.util.ArrayList;
import java.util.List;
```

```
// (1) Component
interface Graphic {
    void paint();
}

// (2) Composite
class Shape implements Graphic {
    List<Graphic> sides = new ArrayList<>();
    public Shape addSide(Graphic side) {
        sides.add(side);
        return this;
    }

    public void removeSide(Graphic side) {
        sides.remove(side);
    }

    public Graphic getSide(int i) {
        return sides.get(i);
    }

    @Override public void paint() {
        for (Graphic side : sides) side.paint();
    }
}

// (3) Leaf
class Line implements Graphic {
    private String name;
    public Line(String name) {
        this.name = name;
    }

    @Override public void paint() {
        System.out.printf("Line [%s]%n", name);
    }
}
```

```

}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Composite Pattern ---");
        Graphic shape = new Shape()
            .addSide(new Line("left"))
            .addSide(new Line("top"))
            .addSide(new Line("right"))
            .addSide(new Line("bottom"));
        shape.paint();
    }
}

```

Một đối tượng đồ họa (Graphic) phức hợp (Shape) là tổ hợp các đối tượng đồ họa đơn (Line). Để xử lý chúng (paint) như nhau, đưa chúng vào một cấu trúc cây và áp dụng mẫu thiết kế Composite.

2. Liên quan

- Chain of Responsibility: các liên kết đến lớp cha thường được dùng trong Chain of Responsibility.
- Decorator: thường được sử dụng với Composite, lúc đó Component chỉ khai báo operation() trong giao diện, Decorator sẽ mở rộng giao diện bằng cách thêm các tác vụ truy cập đối tượng con (add, remove, getChild).
- Flyweight: cho phép dùng chung các Component, nhưng không tham chiếu đến lớp cha.
- Iterator: thường dùng để duyệt danh sách con trong đối tượng Composite.
- Visitor: xác định các tác vụ và hành vi sẽ "viếng thăm" các lớp Leaf và Composite.

3. Java API

java.awt.Container là một Composite, phương thức add(Component) của nó cho phép thêm vào nó một Component hoặc một Container khác.

Framework junit áp dụng mẫu thiết kế Composite, với giao diện Test (Component), TestCase (Leaf) và TestSuite (Composite). Trong JavaEE, mẫu thiết kế Composite View (lớp Presentation) thường dùng cho các ứng dụng portal, được phát triển từ mẫu thiết kế Composite.

4. Sử dụng

Ta có:

- Một cấu trúc chứa các đối tượng đơn và đối tượng phức hợp.

Ta muốn:

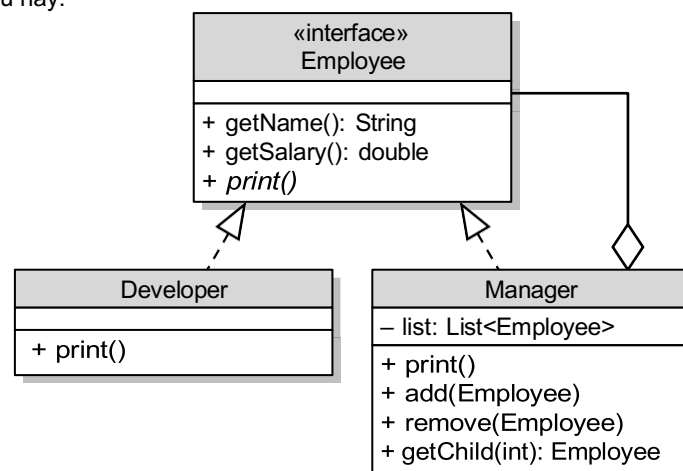
- Client bỏ qua tất cả những khác biệt cơ bản giữa các đối tượng đơn và đối tượng phức hợp.
- Đối xử thống nhất cả các đối tượng đơn lẫn đối tượng phức hợp.

Xem xét sử dụng:

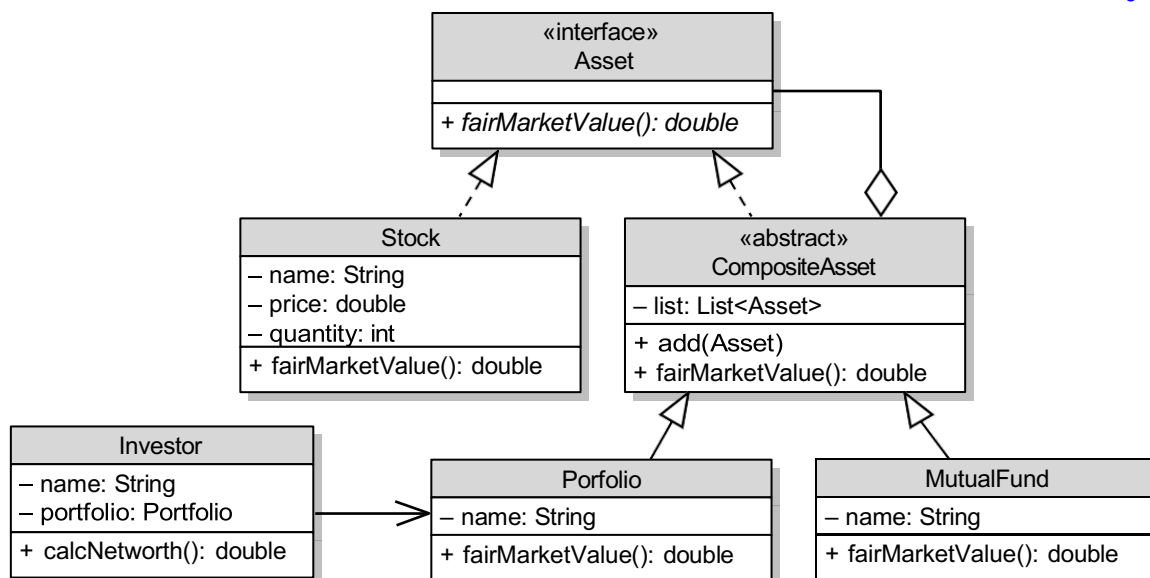
- Decorator, cung cấp các tác vụ mở rộng như thêm đối tượng, loại bỏ đối tượng và tìm kiếm đối tượng.
- Flyweight, để dùng chung trạng thái cho các Component.
- Visitor, xác định các tác vụ được phân phối trên các lớp Leaf và Composite.

5. Bài tập

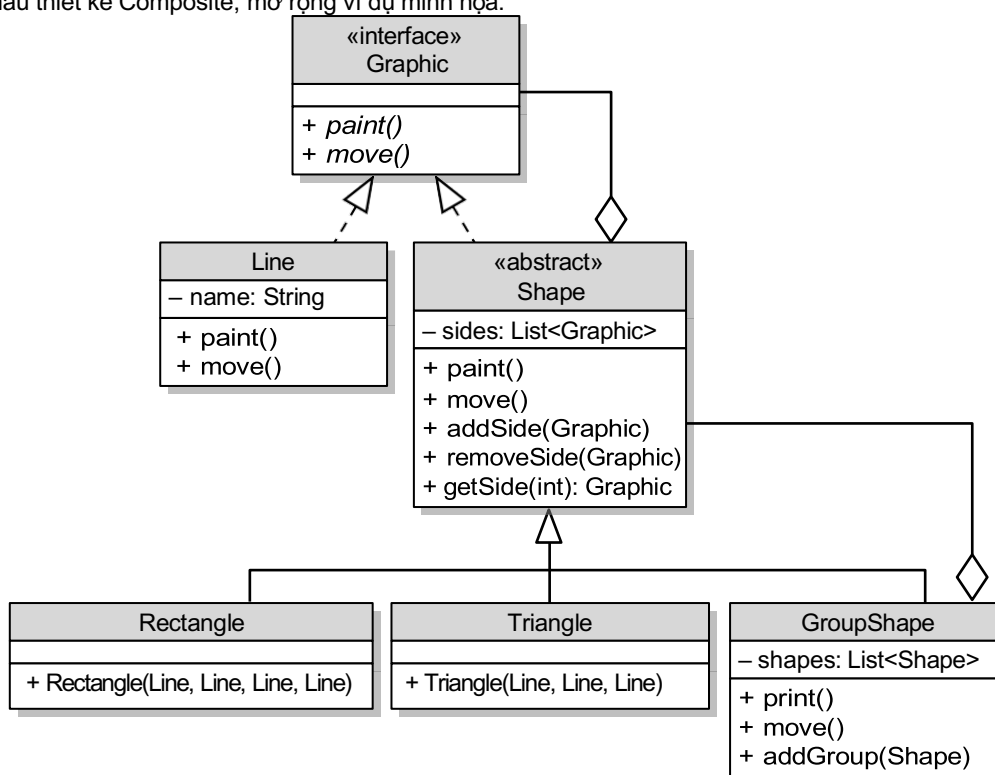
a) Developer và Manager đều là các Employee, nhưng Manager quản lý một danh sách (có thể rỗng) các Employee khác, bao gồm các Developer và các Manager khác dưới quyền. Với Developer, phương thức print() chỉ in ra thông tin cơ bản. Với Manager, ngoài thông tin cơ bản, phương thức print() còn in thông tin về các Employee do Manager quản lý. Áp dụng mẫu thiết kế Composite để thực hiện điều này.



b) Một hệ thống quản lý danh mục đầu tư (Portfolio), bao gồm Stock (cổ phiếu) và Mutual Fund (quỹ đầu tư mở), một Mutual Fund là một danh sách các Stock. Ta muốn áp dụng một giao diện chung cho cả Stock và Mutual Fund để đơn giản hóa việc xử lý. Điều này cho phép thực hiện các hoạt động như tính toán giá trị thực sự của tài sản (fair market value), hoặc mua, bán các loại tài sản. Hãy áp dụng mẫu thiết kế Composite để làm giảm sự phức tạp của việc xây dựng các hoạt động này.



c) Cài đặt cho mẫu thiết kế Composite, mở rộng ví dụ minh họa.



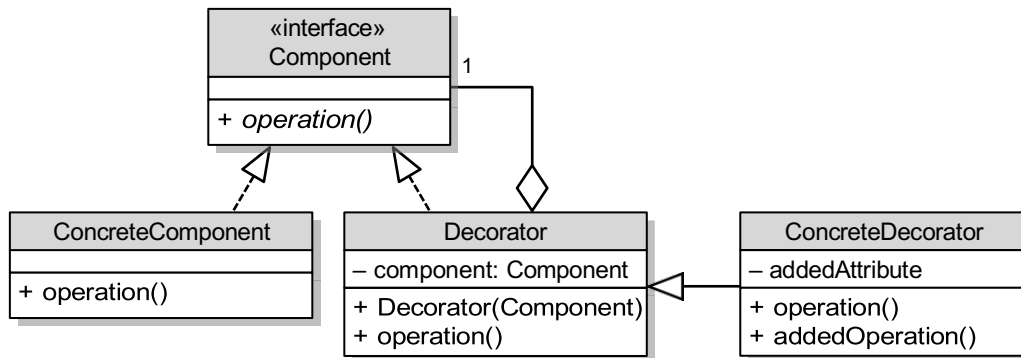
Decorator

Dynamically add responsibility 

Mẫu thiết kế Decorator bổ sung thêm tác vụ cho một đối tượng, không theo chiến lược thừa kế để mở rộng thêm chức năng mà dùng phương pháp "bao bọc" đối tượng gốc lại để "trang trí" thêm chức năng. Điều này cho phép các chức năng bổ sung có thể được loại bỏ khi không cần, hoặc được thêm vào theo một thứ tự khác. Do đặc điểm này, Decorator còn gọi là Wrapper giống như mẫu thiết kế Adapter.

Ý tưởng là cho phép phát triển tiếp code, không cần thay đổi trên code gốc mà vẫn đáp ứng được yêu cầu thay đổi, đảm bảo nguyên tắc OCP.

1. Cài đặt



- Component: định nghĩa giao diện của đối tượng mà ta muốn thêm tác vụ đến nó một cách động.

- ConcreteComponent: định nghĩa đối tượng cụ thể ta muốn thêm tác vụ đến nó.

- Decorator: giữ một tham chiếu tới một đối tượng Component, định nghĩa giao diện phù hợp với giao diện của Component.

Chú ý đặc điểm constructor của nó nhận đối số là đối tượng được "bao bọc".

Khả năng lồng Decorator này vào Decorator khác cho phép bạn xây dựng nên các cấu trúc động.

- ConcreteDecorator: ủy nhiệm lời gọi operation() đến đối tượng Component trong nó và gọi tác vụ bổ sung addedOperation().

// (1) Component

```
interface Component {
    void draw();
}
```

// (2) ConcreteComponent

```
class Window implements Component {
    @Override public void draw() {
        System.out.println("Draw window");
    }
}
```

// (3) Decorator

```
class Decorator implements Component {
    private Component component;

    public Decorator(Component component) {
        this.component = component;
    }

    @Override public void draw() {
        component.draw();
    }
}
```

// (4) ConcreteDecorator

```
class ScrollbarWindow extends Decorator {
    private String scrollbar = "scrollbar";
    public ScrollbarWindow(Component component) {
        super(component);
    }

    @Override public void draw() {
        super.draw();
        System.out.println("Draw " + scrollbar);
    }
}
```

```
class IconWindow extends Decorator {
    private String icon = "icon";
    public IconWindow(Component component) {
```



```

    super(component);
}

@Override public void draw() {
    super.draw();
    System.out.println("Draw " + icon);
}
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Decorator Pattern ---");
        Component iconWindow = new IconWindow(new ScrollbarWindow(new Window()));
        iconWindow.draw();
    }
}

```

2. Liên quan

- Adapter: Decorator cũng gọi là Wrapper như Adapter, nhưng chúng có mục đích khác nhau. Adapter bao đối tượng để thay đổi giao diện của chúng, trong lúc Decorater bao đối tượng để bổ sung thêm hành vi cho nó.
- Composite: Decorator cũng có thể coi như một Composite bị thoái hoá với duy nhất một thành phần (Leaf). Tuy nhiên Decorator không có mục tiêu tạo đối tượng phức hợp (Composite), nó chỉ thêm chức năng cho đối tượng.
- Strategy: Decorator thay đổi bên ngoài của đối tượng, trong lúc Strategy thay đổi bên trong của đối tượng.
- Factory Method: đôi khi được dùng với Decorator để đóng gói các thủ tục thiết lập.

// dùng interface và các lớp trong ví dụ trên

```

interface Creator {
    Component createWindow();
}

class ScrollbarWindowCreator implements Creator {
    @Override public Component createWindow() {
        return new ScrollbarWindow(new Window());
    }
}

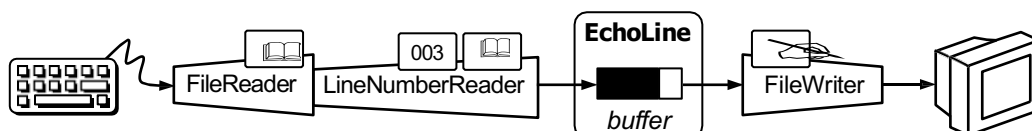
class IconWindowCreator implements Creator {
    @Override public Component createWindow() {
        return new IconWindow(new ScrollbarWindow(new Window()));
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Decorator + Factory Method ---");
        Creator[] creators = { new ScrollbarWindowCreator(), new IconWindowCreator() };
        for (Creator creator : creators) {
            creator.createWindow().draw();
        }
    }
}

```

3. Java API

Stream API (java.io.InputStream, OutputStream, Reader và Writer) được thiết kế theo mẫu thiết kế Decorator. Điều này cho phép "lồng stream": gọi constructor của stream có tính năng tăng cường với đối số là đối tượng stream có tính năng cơ bản, nhằm tăng tính năng của stream cơ bản. Thường các stream được lồng nhau cho đến khi nhận được stream có phương thức phù hợp với nhu cầu sử dụng.



```

import java.io.FileDescriptor;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.LineNumberReader;
// ...
FileReader in = new FileReader(FileDescriptor.in);
FileWriter out = new FileWriter(FileDescriptor.out);
// lồng stream, stream mới "trang trí" thêm tính năng đánh số cho dòng
LineNumberReader lineIn = new LineNumberReader(in);

```

```

char[] buffer = new char[256];
int numberRead;
while ((numberRead = lineIn.read(buffer)) > -1) {
    String upper = new String(buffer, 0, numberRead).toUpperCase();
    out.write(lineIn.getLineNumber() + ": " + upper);
}

```

4. Sử dụng

Ta có:

- Lớp Component nhưng không dễ thừa kế nó.

Ta muốn:

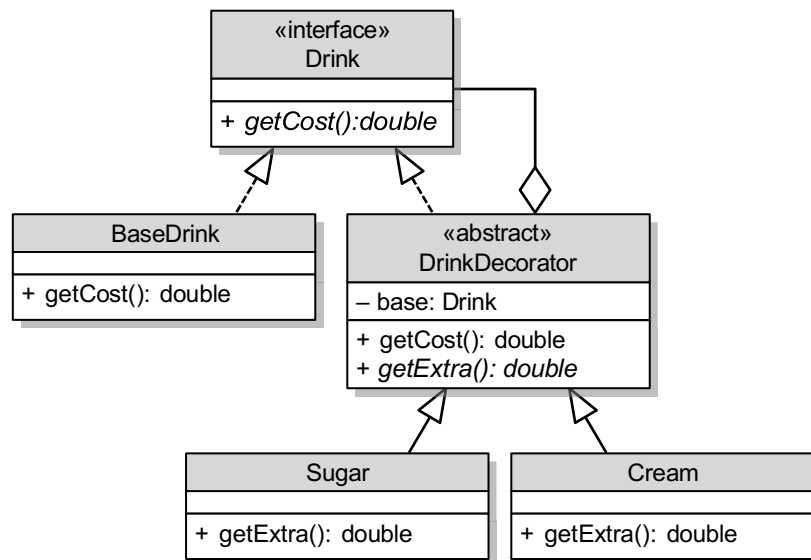
- Bổ sung thuộc tính hoặc hành vi cho một đối tượng một cách động.
- Thay đổi một số đối tượng trong một lớp mà không ảnh hưởng đến đối tượng khác.
- Tránh thừa kế vì có thể dẫn đến có quá nhiều lớp.

Xem xét sử dụng:

- Adapter, cho phép thiết lập một giao diện giữa các lớp khác nhau.
- Composite, tạo đối tượng phức hợp mà cũng không cần mở rộng giao diện.
- Proxy, cho phép điều khiển truy cập đến đối tượng nó đại diện.
- Strategy, làm thay đổi đối tượng mà không cần "bao bọc" nó.

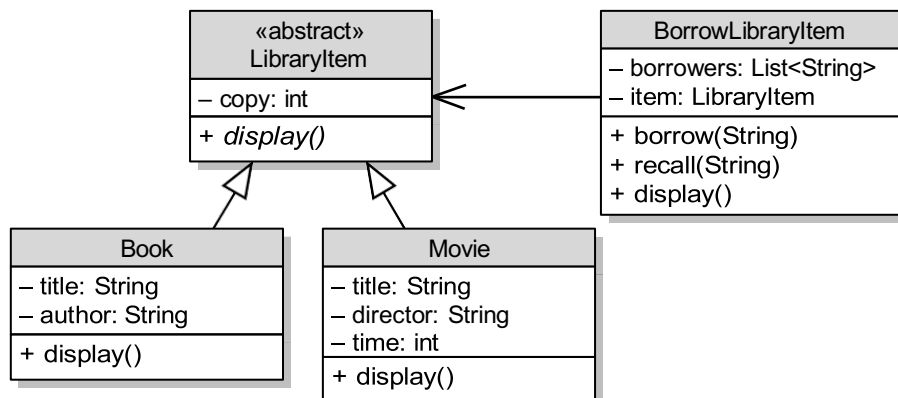
5. Bài tập

a) Cửa hàng cung cấp thức uống (BaseDrink) với giá cơ bản là \$1.0. Khách hàng có thể tùy chọn thêm đường (Sugar) với phụ phí \$0.5, thêm kem (Cream) với phụ phí \$0.25, hoặc thêm cả hai. Tổng giá được tính tùy theo "thiết kế" thức uống của khách. Bạn hãy áp dụng mẫu thiết kế Decorator, đáp ứng yêu cầu này.



b) Tài nguyên của thư viện (LibraryItem) gồm hai loại Book và Movie, để xây dựng chương trình quản lý thư viện, ta cần mở rộng lớp LibraryItem thành lớp BorrowLibraryItem, thêm vào đó danh sách độc giả mượn tài nguyên và thêm hai phương thức mới: borrow (cho mượn), recall (thu hồi).

Bạn hãy áp dụng mẫu thiết kế Decorator để viết lớp BorrowLibraryItem, đáp ứng yêu cầu này.



Facade

Interface for subsystem

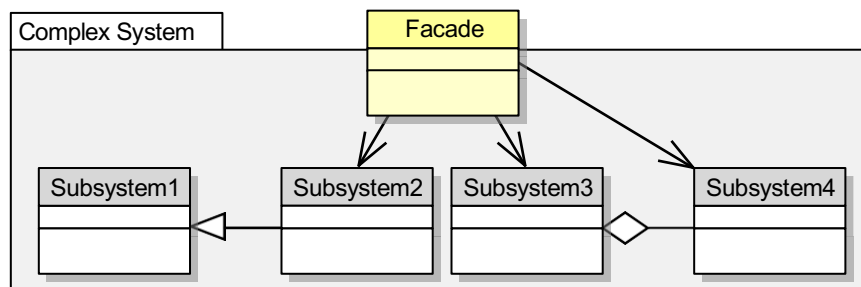
Mẫu thiết kế Facade (façade [fə'sɔ:d]: mặt tiền) cung cấp *một giao diện chung đơn giản* thay cho một nhóm các giao diện có trong một hệ thống con (subsystem²). Mẫu thiết kế Facade định nghĩa một giao diện cấp cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này vì chỉ cần giao tiếp với một giao diện chung duy nhất.

Mẫu thiết kế Facade cho phép các đối tượng truy cập vào hệ thống con bằng cách sử dụng giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp trong hệ thống con.

Việc sử dụng mẫu thiết kế Facade đem lại các lợi ích sau:

- Người dùng không cần biết đến sự phức tạp bên trong hệ thống con mà dễ dàng sử dụng hệ thống con vì chỉ giao tiếp với hệ thống con thông qua một giao diện chung đơn giản³.
- Chú ý là mẫu thiết kế Facade không "đóng gói" (encapsulation) hệ thống con theo nghĩa hạn chế truy xuất; nó cung cấp một giao diện đơn giản trong lúc vẫn bộc lộ đầy đủ các chức năng của hệ thống con. Những người dùng cao cấp vẫn có thể truy xuất vào sâu bên trong hệ thống con khi cần thiết, chẳng hạn để sửa chữa nâng cấp hệ thống con.
- Nâng cao khả năng độc lập của hệ thống con do cho phép nâng cấp đơn thể trong hệ thống con mà không cần phải sửa lại mã lệnh từ phía người dùng.
- Giúp phân lớp (layer) hệ thống con và phân nhóm sự phụ thuộc của các đối tượng trong hệ thống con.

1. Cài đặt



- Facade: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của client, sẽ ủy nhiệm việc thực hiện đến các đối tượng của hệ thống con tương ứng. Đôi khi đối tượng Facade được gọi là "God Object".

- Các lớp Subsystem: cài đặt các chức năng của hệ thống con, phối hợp xử lý các công việc được Facade bọc lộ ra bên ngoài. Các lớp này không cần biết Facade và không tham chiếu đến nó.

Hai hệ thống con của hai lớp Facade khác nhau có thể có những lớp chung.

```
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
```

```
// (1) Subsystem
class Student {
    int code;
    String name;
    public Student(int code, String name) {
        this.code = code;
        this.name = name;
    }

    @Override public String toString() {
        return String.format("[%d] %s", code, name);
    }
}

class Course {
    int code;
    String name;
    public Course(int code, String name) {
        this.code = code;
        this.name = name;
    }
}

class Section {
    String name;
    Course course;
    private List<Student> students = new LinkedList<>();
    public Section(String name) { this.name = name; }
    public void addStudent(Student student) { students.add(student); }
```

² Subsystem là nhóm các lớp, hoặc nhóm các lớp với các subsystem khác; chúng cộng tác với nhau để thực hiện một số tác vụ.

³ Mỗi hệ thống con có thể có nhiều giao diện Facade.

```

    public List<Student> getStudents() { return students; }
}

class Campus {
    static HashMap<Integer, Student> students = new HashMap();
    static HashMap<Integer, Course> courses = new HashMap();

    public static void setStudent(int studentCode, String studentName) {
        students.put(studentCode, new Student(studentCode, studentName));
    }

    public static Student getStudent(int studentCode) {
        return students.get(studentCode);
    }

    public static void setCourse(int courseCode, String courseName) {
        courses.put(courseCode, new Course(courseCode, courseName));
    }

    public static Course getCourse(int courseCode) {
        return courses.get(courseCode);
    }
}

// (2) Facade
class Facade {
    private static HashMap<String, Section> sections = new HashMap();
    public static void buildCampus() {
        Campus.setCourse(1000, "Operating System");
        Campus.setCourse(2000, "Core Java");
        Campus.setStudent(100, "Bill Gates");
        Campus.setStudent(101, "James Gosling");
        Campus.setStudent(102, "Linus Tovarld");
    }

    public static void buildSection(String sectionName, int courseCode) {
        Section section = new Section(sectionName);
        section.course = Campus.getCourse(courseCode);
        sections.put(sectionName, section);
    }

    public void enroll(String sectionName, int... studentCode) {
        for (int code : studentCode)
            sections.get(sectionName).addStudent(Campus.getStudent(code));
    }

    public void display(String sectionName) {
        Section section = sections.get(sectionName);
        String sName = section.name;
        String cName = section.course.name;
        List<Student> students = section.getStudents();
        System.out.printf("%nCourse Name: %s [%s]%n", cName, sName);
        System.out.println("Student List: ");
        for (Student student : students)
            System.out.println(" " + student);
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Facade Pattern ---");
        Facade facade = new Facade();
        Facade.buildCampus();
        Facade.buildSection("OS1000", 1000);
        Facade.buildSection("CJ2000", 2000);
        facade.enroll("OS1000", 100, 102);
        facade.enroll("CJ2000", 101, 100);
        facade.display("OS1000");
        facade.display("CJ2000");
    }
}

```

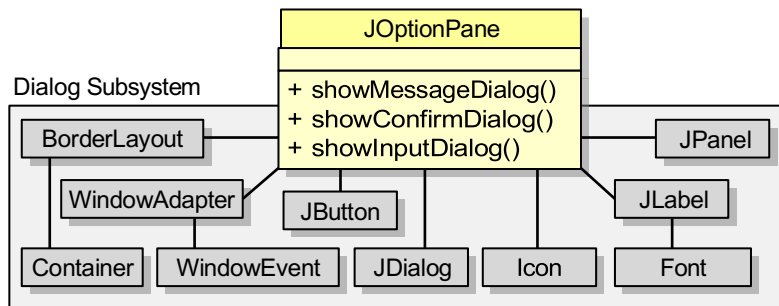
2. Liên quan

- Abstract Factory: thường dùng để tạo giao diện cho một hệ thống con một cách độc lập, có thể dùng như một Facade.
- Singleton: đối tượng Facade thường là một Singleton vì chỉ cần một đối tượng Facade.
- Mediator: tương tự như Facade, "bao bọc" một hệ thống con làm cho nó dễ sử dụng hơn. Nhưng Facade không định nghĩa chức năng mới cho hệ thống con, Facade chỉ giúp đơn giản hóa việc truy cập vào các lớp ủy nhiệm. Lớp Facade cũng không được các lớp của hệ thống con biết đến.
- Adapter: Facade có thể được xem như Adapter cho một hệ thống con. Trong trường hợp lời gọi đến hệ thống con quá phức tạp hoặc không phù hợp, lớp Facade "bao bọc" toàn bộ hệ thống con và cung cấp giao diện đơn giản hơn.

3. Java API

JDBC trong Java là một ví dụ tốt cho mẫu thiết kế Facade, nó đem đến một khung công việc (framework) dễ dùng khi truy xuất cơ sở dữ liệu. Trong JavaEE, mẫu thiết kế Session Facade (lớp Business) được phát triển từ mẫu thiết kế Facade.

JOptionPane là một lớp Facade, nó đơn giản hóa việc truy cập hệ thống các lớp giúp hiển thị các loại hộp thoại khác nhau.



4. Sử dụng

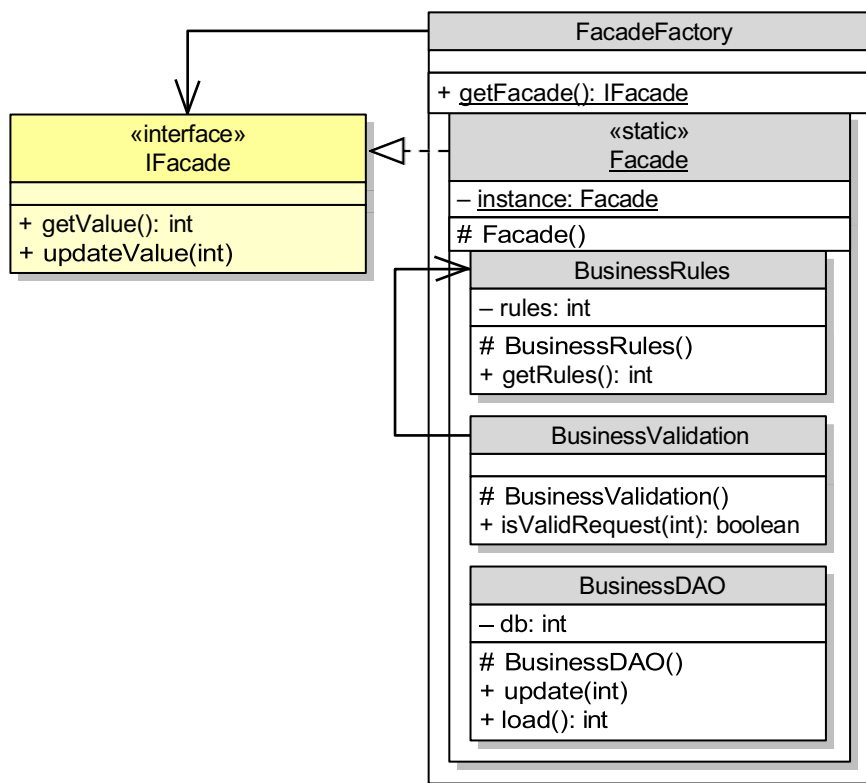
Ta có:

- Các hệ thống con có phần trừu tượng và phần hiện thực liên kết chặt chẽ.
- Hệ thống tiến hóa và trở nên phức tạp hơn, nhưng vẫn phải giữ giao diện giao tiếp đơn giản.

Ta muốn:

- Phân cấp giao diện cho người dùng, người dùng phân quyền khác nhau có giao diện khác nhau.
- Phân lớp hệ thống với mỗi lớp có điểm nhập xác định.
- Che giấu các tác vụ của hệ thống con, chỉ có thể gọi chúng thông qua giao diện Facade.

5. Bài tập



Tác vụ lưu một trị vào cơ sở dữ liệu được thực hiện phức tạp với các hệ thống con:

- Từ lớp BusinessRules, lấy trị rules bằng phương thức getRules().
- Trong lớp BusinessValidation, kiểm tra tính hợp lệ của trị value muốn lưu bằng phương thức isValidRequest(). Giả sử luật dùng lưu trị đơn giản: value < rules.
- Nếu trị muốn lưu hợp lệ, dùng phương thức update() của lớp BusinessDAO, lưu trị vào cơ sở dữ liệu. Có thể thực hiện đơn giản bằng cách lưu vào trị db của BusinessDAO.

Tác vụ lấy một trị từ cơ sở dữ liệu (từ trị db) phải gọi phương thức load() của lớp BusinessDAO.

Để cung cấp giao diện đơn giản cho người dùng, áp dụng mẫu thiết kế Facade:

- Dùng lớp nội (inner class) để che giấu các hệ thống con: các lớp BusinessRules, BusinessValidation và BusinessDAO là lớp nội của Facade; Facade là lớp nội của FacadeFactory.
 - Phương thức factory getFacade() sẽ cung cấp đối tượng Singleton Facade, cài đặt giao diện IFacade cho người dùng.
- Hãy hiện thực yêu cầu trên.

Flyweight

Efficiently share objects 

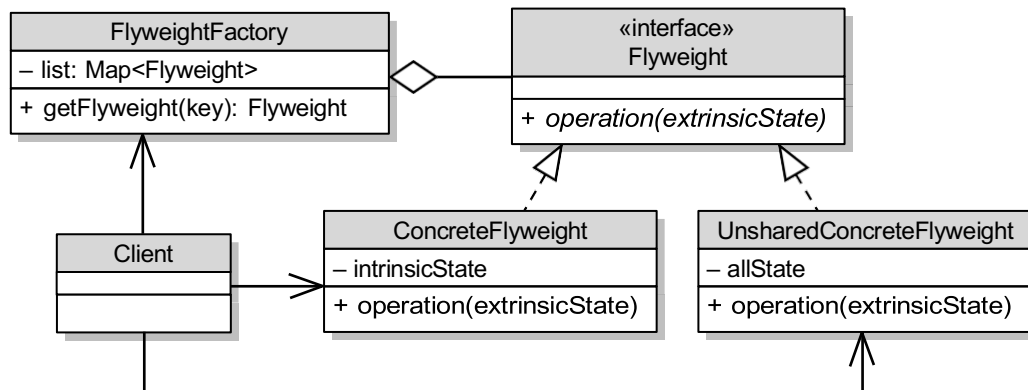
Trạng thái của một đối tượng có thể chứa một trong hoặc cả hai loại⁴ sau:

- Trạng thái bên trong (intrinsic): độc lập với ngữ cảnh của đối tượng, trạng thái chung (sharing) cho tất cả (hoặc một nhóm) đối tượng của một lớp. Ví dụ, thông tin về công ty trên danh thiếp của tất cả nhân viên thuộc công ty là giống nhau.
- Trạng thái bên ngoài (extrinsic) là phụ thuộc và biến đổi theo ngữ cảnh của đối tượng, trạng thái đặc thù cho từng đối tượng thuộc lớp. Ví dụ, tên và số điện thoại của nhân viên trên danh thiếp là khác nhau dù họ chung một công ty.

Mẫu thiết kế Flyweight đề nghị tách trạng thái intrinsic và đóng gói vào một đối tượng riêng gọi là Flyweight. Nhóm có số lượng lớn các đối tượng có thể dùng chung đối tượng Flyweight này để thể hiện phần trạng thái intrinsic của chúng, dẫn đến thu giảm yêu cầu lưu trữ.

Như vậy, trạng thái intrinsic được lấy từ thực thể dùng chung, trạng thái extrinsic được truyền như tham số.

1. Cài đặt



- FlyweightFactory: bảo đảm tạo và lưu trữ các Flyweight khác nhau. FlyweightFactory được thiết kế như một Singleton. Khi Client cần đến thực thể Flyweight, nó gọi phương thức getFlyweight() của FlyweightFactory với tham số là kiểu của Flyweight yêu cầu.

- Flyweight: giao diện giúp ConcreteFlyweight có thể thao tác với trạng thái extrinsic khi cần.

- ConcreteFlyweight: chứa trạng thái intrinsic của đối tượng, thường được thiết kế như lớp nội (inner class) của FlyweightFactory với constructor là private để ngăn Client tạo trực tiếp đối tượng Flyweight.

- UnsharedConcreteFlyweight: không phải lớp cài đặt nào của Flyweight đều dùng chung, lớp không dùng chung này bổ sung cho cây dẫn xuất từ Flyweight. Thường hiếm cài đặt.

```
import java.util.HashMap;
import java.util.Map;
```

```
// (1) Flyweight
```

```
interface Letter {
    String getValue();
}
```

```
// (2) FlyweightFactory
```

```
class LetterFactory {
    public static LetterFactory factory;
    public static Map<String, Letter> list = new HashMap<>();

    private LetterFactory() {}

    public static synchronized LetterFactory getFactory() {
        if (factory == null) factory = new LetterFactory();
        return factory;
    }

    public Letter getLetter(String key) {
        if (!list.containsKey(key)) {
            list.put(key, new Char(key));
        }
        return list.get(key);
    }
}
```

```
// (3) ConcreteFlyweight
```

```
class Char implements Letter {
    private String value;
    private Char(String value) {
        System.out.println("create letter: " + value);
        this.value = value;
    }
}
```

⁴ Có tài liệu phân thành ba loại: unshared (tương đương extrinsic), intrinsic, extrinsic (trạng thái được tính trong thời gian chạy).

```

    }

    @Override public String getValue() {
        return value;
    }
}

class WordProcessor {
    private StringBuilder sb = new StringBuilder();
    public void add(Letter letter) {
        sb.append(letter.getValue());
    }

    public void print() {
        System.out.println(sb.toString());
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Flyweight Pattern ---");
        WordProcessor processor = new WordProcessor();
        String s = "Google is good";
        for (int i = 0; i < s.length(); ++i) {
            String value = s.substring(i, i + 1);
            processor.add(LetterFactory.getFactory().getLetter(value));
        }
        processor.print();
    }
}

```

WordProcessor làm việc không hiệu quả với nhiều Letter. Bạn có thể thấy điều này nếu đưa lớp Char ra ngoài và viết lại lệnh: `processor.add(new Char(value));`

Các Letter có trạng thái intrinsic là value, có thể dùng chung nếu chúng có value giống nhau. Đóng gói trạng thái intrinsic này vào đối tượng Flyweight để giảm số đối tượng cần tạo. Trạng thái extrinsic, ví dụ số lần ký tự đó xuất hiện, không mô tả ở đây.

2. Liên quan

- Composite: nên cài đặt Flyweight kết hợp Composite để tạo nên các cấu trúc phân cấp có các node dùng chung.
- State và Strategy: nên cài đặt các đối tượng State và Strategy như là các Flyweight.

3. Java API

`java.lang.String`.

4. Sử dụng

Ta có:

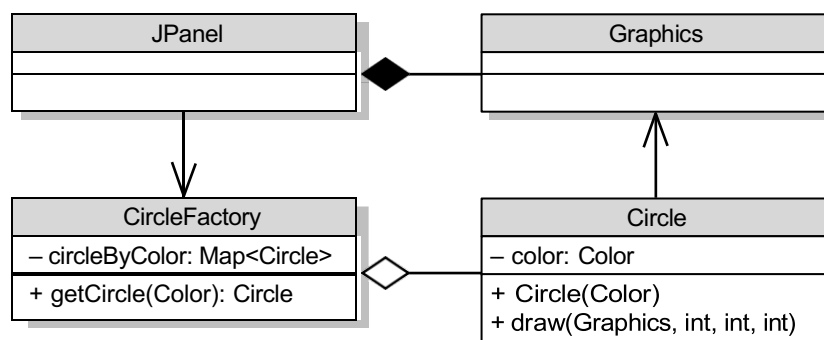
- Chương trình sử dụng một số lớn đối tượng trong bộ nhớ. Cần giảm số đối tượng này.
- Nhóm các đối tượng có một số trạng thái dùng chung.

Ta muốn:

- Cài đặt một hệ thống mà việc sử dụng bộ nhớ bị hạn chế.

5. Bài tập

Viết chương trình vẽ 1000 hình tròn (Circle) lên một JPanel. Các hình tròn có tâm và bán kính khác nhau, được vẽ bằng 6 màu: red, blue, yellow, orange, black và white. Nếu áp dụng mẫu thiết kế Flyweight, thay vì phải tạo 1000 đối tượng Circle, CircleFactory chỉ tạo 6 đối tượng Circle. Hãy thực hiện yêu cầu này.



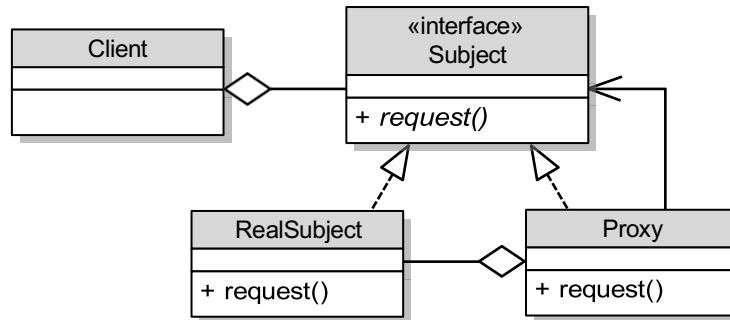
Proxy

Placeholder for objects 

Mẫu thiết kế Proxy cung cấp một *đối tượng đại diện* (Proxy) điều khiển việc tạo ra và truy cập đến một *đối tượng thực* khác (RealSubject). Proxy thường là một đối tượng nhỏ đứng chắn trước đối tượng phức tạp hơn, Proxy chỉ kích hoạt đối tượng phức tạp này khi đạt được một số điều kiện nhất định.

Ý tưởng của mẫu thiết kế Proxy là cung cấp một đối tượng thay thế (surrogate) hoặc giữ chỗ (placeholder) cho một đối tượng thực. Đối tượng thực có thể chưa có sẵn (nằm trên máy ở xa, chưa nạp vào, đã chuyển ra đĩa (swap out)) hoặc cần kiểm soát được việc truy cập vào nó (authentication, audit, preprocessing, logging, transaction context setup). Nói cách khác, con đường truy cập đối tượng thực (RealSubject) buộc phải thông qua Proxy.

1. Cài đặt



- Subject: giao diện chung cho RealSubject lẫn Proxy, để Proxy có mặt mọi nơi mà RealSubject được quan tâm, đại diện được cho RealSubject.

Tùy nhiệm vụ, có nhiều loại Proxy:

- + Virtual proxy: xử lý việc tạo RealSubject dựa trên thông tin khác (ví dụ từ tên tập tin), vì vậy có trì hoãn khi tạo RealSubject.
- + Protection proxy: khi gọi yêu cầu, phải vượt qua xác thực rồi mới tạo RealSubject.
- + Remote proxy: mã hóa yêu cầu tạo đối tượng và gửi chúng qua mạng, RealSubject sẽ được tạo trong một không gian địa chỉ khác.
- + Smart proxy: thêm yêu cầu hoặc thay đổi yêu cầu trước khi gửi yêu cầu để tạo hoặc truy cập đối tượng.

- RealSubject: lớp mà Proxy sẽ đại diện.

- Proxy: lớp được dùng để tạo ra, điều khiển, tăng cường, xác thực truy cập đến RealSubject. Proxy giữ một tham chiếu đến RealSubject để ủy nhiệm các lời gọi cho RealSubject khi đạt điều kiện gọi.

// (1) Subject

```
interface Subject {
    String request();
}
```

// (2) RealSubject

```
class RealSubject implements Subject {
    @Override public String request() {
        return "RealSubject::request()";
    }
}
```

// (3) Proxy

```
class VirtualProxy implements Subject {
    Subject subject = null;
    @Override public String request() {
        if (subject == null) {
            System.out.println("Virtual Proxy: Subject inactive");
            subject = new RealSubject();
        }
        System.out.println("Virtual Proxy: Subject active");
        return "Virtual Proxy: " + subject.request();
    }
}
```

```
class ProtectionProxy implements Subject {
    Subject subject = null;
    private final String PASSWORD = "s3kr3t";
    public String authenticate(String password) {
        if (!this.password.equals(PASSWORD)) {
            return "Protection Proxy: Access denied";
        } else {
            subject = new RealSubject();
            return "Protection Proxy: Access granted";
        }
    }
}
```

```

    }

    @Override public String request() {
        if (subject == null) {
            return "Protection Proxy: Authenticate first";
        } else {
            return "Protection Proxy: " + subject.request();
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("---- Proxy Pattern ----");
        Subject subject = new VirtualProxy();
        System.out.println(subject.request());
        System.out.println();
        Subject pSubject = new ProtectionProxy();
        System.out.println(pSubject.request());
        System.out.println(((ProtectionProxy)pSubject).authenticate("password"));
        System.out.println(((ProtectionProxy)pSubject).authenticate("s3kr3t"));
        System.out.println(pSubject.request());
    }
}

```

2. Liên quan

- Adapter: Adapter cung cấp một giao diện khác cho đối tượng mà nó tiếp hợp. Trong lúc Proxy cung cấp giao diện giống với Subject mà nó đại diện. Nhiệm vụ hai mẫu thiết kế này khác nhau.
- Decorator: Proxy cài đặt giống như Decorator nhưng có nhiệm vụ khác.

3. Java API

Mẫu thiết kế Proxy đã được tích hợp vào Java API giúp người dùng dễ dàng tạo một Proxy cho đối tượng RealSubject cần kiểm soát truy cập. Để áp dụng mẫu thiết kế này, bạn tạo lớp xử lý cho proxy (ProxyHandler):

- Trong lớp này, khi đạt điều kiện tạo đối tượng, tạo đối tượng RealSubject để ủy nhiệm các lời gọi cho RealSubject.
- Phương thức invoke(): cài đặt từ giao diện java.lang.reflect.InvocationHandler. Trong phương thức này, chọn phương thức gọi và ủy nhiệm lời gọi phương thức cho đối tượng RealSubject thực hiện.

Object invoke(Object proxy, Method method, Object[] args);

proxy: đối tượng proxy do Proxy.newProxyInstance() trả về.

method: đóng gói phương thức triệu gọi, dùng phương thức getName() để xác định phương thức cần gọi.

args: đối số của phương thức triệu gọi, định nghĩa tại giao diện Subject.

- Phương thức createProxy(): tạo đối tượng proxy bằng phương thức static newProxyInstance() của lớp Proxy. Các đối số của phương thức newProxyInstance() dùng cơ chế reflection của Java để lấy thông tin từ Subject, giao diện của đối tượng RealSubject mà proxy đại diện.

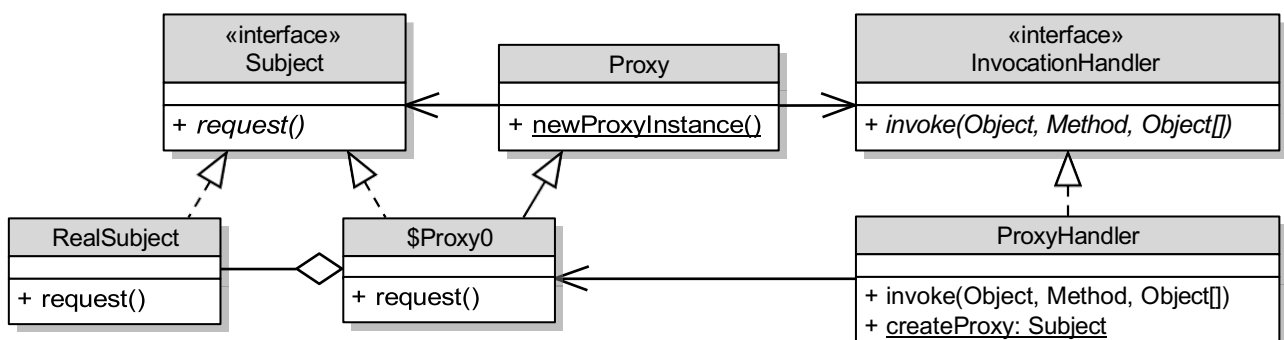
static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler);

loader: bộ nạp lớp của Subject, Subject.class.getClassLoader() hoặc subject.getClass().getClassLoader().

interfaces: interface của Subject, new Class<?>[] { Subject.class } hoặc subject.getClass().getInterfaces().

handler: lớp xử lý new ProxyHandler().

newProxyInstance() sẽ trả về đối tượng vừa dẫn xuất lớp java.lang.reflect.Proxy, vừa cài đặt giao diện Subject.



```

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

```

```

// (1) Subject
interface Subject {
    String request();
}

```

```
// (2) RealSubject
class RealSubject implements Subject {
    @Override public String request() {
        return "RealSubject::request()";
    }
}

// (3) Proxy Handler
class ProxyHandler implements InvocationHandler {
    private Subject subject = null;
    private final String PASSWORD = "s3kr3t";
    private ProxyHandler(String password) {
        if (!password.equals(PASSWORD)) {
            System.out.println("Protection Proxy: Access denied");
        } else {
            subject = new RealSubject();
            System.out.println("Protection Proxy: Access granted");
        }
    }

    public static Subject createProxy(String password) {
        return (Subject)Proxy.newProxyInstance(Subject.class.getClassLoader(),
            new Class<?>[]{ Subject.class }, new ProxyHandler(password));
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws IllegalAccessException {
        try {
            if (method.getName().equals("request")) return method.invoke(subject, args);
            else throw new IllegalAccessException();
        } catch (InvocationTargetException e) {
            e.printStackTrace(System.err);
        }
        return null;
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Proxy Pattern in Java API ---");
        Subject pSubject = ProxyHandler.createProxy("s13kr3t");
        try {
            System.out.println(pSubject.request());
        } catch (NullPointerException e) {
            System.err.println("Protection Proxy: Authenticate first");
        }
    }
}
```

4. Sử dụng

Ta có:

- Đối tượng tốn nhiều chi phí để tạo.
- Đối tượng cần xác thực để truy cập.
- Đối tượng truy cập từ xa.
- Đối tượng cần thực hiện một số hành động trước khi truy cập.

Ta muốn:

- Tạo đối tượng chỉ khi có yêu cầu đến tác vụ của chúng.
- Thực hiện việc kiểm tra hoặc dọn dẹp khi truy cập đến đối tượng.
- Một đối tượng cục bộ truy cập đến một đối tượng từ xa.
- Cài đặt các quyền truy cập lên đối tượng khi yêu cầu đến các tác vụ của đối tượng đó.

5. Bài tập

Lớp RealProduct cài đặt giao diện Product như hình bên. Áp dụng framework Proxy của Java API, tạo hai lớp xử lý:

- EmployeeHandler tạo đối tượng proxy chỉ gọi được các phương thức getters của Product.
- ManagerHandler tạo đối tượng proxy gọi được tất cả các phương thức của Product và có ghi nhận vào tập tin log khi gọi phương thức.

«interface» Product
+ getName(): String + getPrice(): double + setName(String) + setPrice(): double

Behavioral

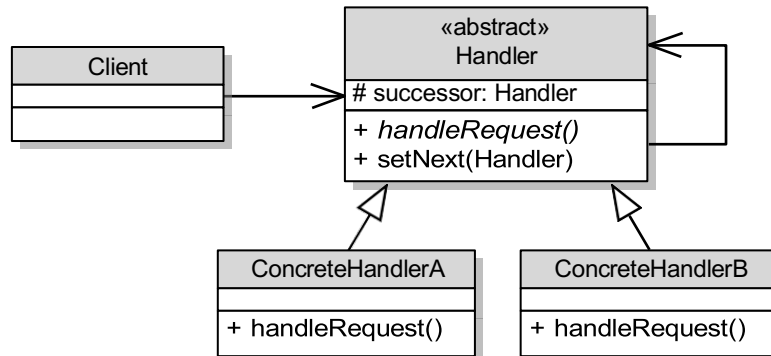
Chain of Responsibility

Decouple sender and receiver 

Mẫu thiết kế Chain of Responsibility giúp tránh kết nối trực tiếp, quá chặt giữa đối tượng gửi yêu cầu và đối tượng nhận yêu cầu, khi yêu cầu được xử lý bởi một hoặc nhiều đối tượng.

Mẫu thiết kế này *tạo một dây chuyền (chain) các đối tượng nhận yêu cầu*, rồi truyền yêu cầu theo dây chuyền đó. Các đối tượng nhận yêu cầu để xử lý có thể là toàn bộ dây chuyền, một phần dây chuyền hoặc chỉ là một đối tượng trong dây chuyền. Bằng cách này, đối tượng gửi yêu cầu không cần biết yêu cầu sẽ được xử lý bởi đối tượng nào.

1. Cài đặt



- Handler: định nghĩa giao diện chung để xử lý yêu cầu. Chứa tham chiếu đến đối tượng xử lý yêu cầu ngay sau nó, gọi là successor. Phương thức setNext() dùng thiết lập đối tượng successor này, nếu cần chuyển tiếp yêu cầu.

- ConcreteHandler: xử lý yêu cầu với các mức độ khác nhau, nếu không xử lý yêu cầu nó có thể chuyển tiếp yêu cầu đến successor của nó.

// (1) Handler

```

abstract class AbstractLogger {
    enum Level { INFO, DEBUG, ERROR };
    protected Level level;
    protected AbstractLogger nextLogger = null;

    abstract protected void write(String message);

    public void setNextLogger(AbstractLogger nextLogger) {
        this.nextLogger = nextLogger;
    }

    public void logMessage(Level level, String message) {
        if (this.level.compareTo(level) <= 0) write(message);
        if (nextLogger != null) nextLogger.logMessage(level, message);
        else
            System.out.println("--- end of chain ---");
    }
}
  
```

// (2) ConcreteHandler

```

class ConsoleLogger extends AbstractLogger {
    public ConsoleLogger(Level level) {
        this.level = level;
    }

    @Override protected void write(String message) {
        System.out.println("[Standard Console]: " + message);
    }
}
  
```

```

class ErrorLogger extends AbstractLogger {
    public ErrorLogger(Level level) {
        this.level = level;
    }

    @Override protected void write(String message) {
        System.out.println("[Error Console]: " + message);
    }
}
  
```

```

class FileLogger extends AbstractLogger {
    public FileLogger(Level level) {
    }
}
  
```

```

    this.level = level;
}

@Override
protected void write(String message) {
    System.out.println("[Log File]: " + message);
}
}

public class Client {

    private static AbstractLogger setChainOfLoggers() {
        AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.Level.ERROR);
        AbstractLogger fileLogger = new FileLogger(AbstractLogger.Level.DEBUG);
        AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.Level.INFO);
        errorLogger.setNextLogger(fileLogger);
        fileLogger.setNextLogger(consoleLogger);
        return errorLogger;
    }

    public static void main(String[] args) {
        System.out.println("--- Chain of Responsibility Pattern ---");
        AbstractLogger loggerChain = setChainOfLoggers();
        loggerChain.logMessage(AbstractLogger.Level.INFO, "Information level.");
        loggerChain.logMessage(AbstractLogger.Level.DEBUG, "Debug level.");
        loggerChain.logMessage(AbstractLogger.Level.ERROR, "Error level.");
    }
}

```

AbstractLogger định nghĩa chuỗi xử lý "đổ xuống": ERROR → DEBUG → INFO, tùy mức độ lỗi mà toàn chuỗi hoặc một phần chuỗi xử lý ghi nhận lỗi.

2. Liên quan

- Composite: mẫu thiết kế Chain of Responsibility thường kết hợp với mẫu thiết kế Composite, khi đó các thành phần con của một Composite được xem như successor của nó.

3. Java API

Các bước xử lý ngoại lệ (exception) được thực hiện theo mẫu thiết kế Chain of Responsibility.

java.util.logging.Logger với phương thức xử lý log().

HttpServletRequest.RequestDispatcher() trong API của servlet/JSP với phương thức forward().

javax.servlet.Filter với phương thức xử lý doFilter().

4. Sử dụng

Ta có:

- Nhiều hơn một đối tượng xử lý yêu cầu, không chỉ định rõ đối tượng sẽ xử lý yêu cầu.
- Một đối tượng xử lý yêu cầu cần chuyển yêu cầu đến một đối tượng xử lý yêu cầu khác trong một dây chuyền.
- Tập hợp các đối tượng xử lý yêu cầu có thể thay đổi một cách động.

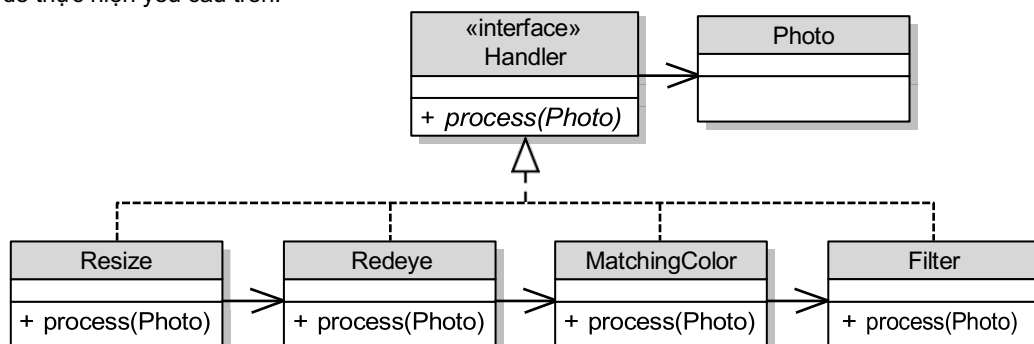
Ta muốn:

- Linh hoạt khi gán các yêu cầu để xử lý.

5. Bài tập

a) Tiến trình xử lý checkout một đơn hàng gồm các bước: CalculatePayment (tính giá trị đơn hàng) và AddCustomerInfo (điền thông tin khách hàng vào đơn hàng). Một bước mới cần đưa vào là SaveOrderInfo (lưu thông tin đơn hàng vào tập tin log). Áp dụng mẫu thiết kế Chain of Responsibility để tiến trình có hai bước như lúc đầu, sau đó hiệu chỉnh để đưa bước thứ ba vào.

b) Chương trình xử lý ảnh cho phép chuỗi xử lý ảnh tự động theo lô: Resize (hiệu chỉnh kích thước) → Redeye (loại bỏ lỗi mắt đỏ) → Color matching (hiệu chỉnh màu) → Filter (áp dụng các bộ lọc định sẵn). Ảnh sẽ được tự động nhận dạng và xử lý tại mỗi khâu, khâu xử lý không cần thiết (ví dụ do ảnh không có lỗi mắt đỏ) có thể được bỏ qua. Áp dụng mẫu thiết kế Chain of Responsibility để thực hiện yêu cầu trên.



Command

Capture actions 

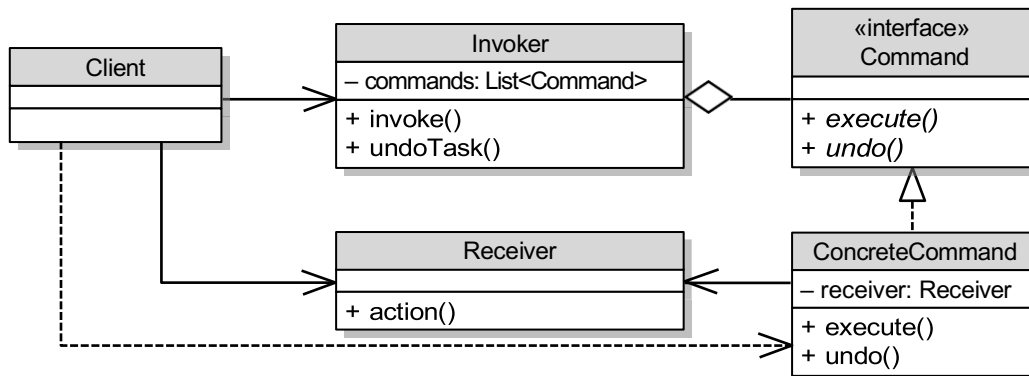
Mẫu thiết kế Command đóng gói yêu cầu (request) như một đối tượng lệnh (Command) rồi truyền đối tượng này cho đối tượng triệu gọi (Invoker) như một tham số. Tùy theo yêu cầu của Client, Invoker triệu gọi đối tượng lệnh thích hợp để thực thi. Đối tượng lệnh ủy nhiệm việc thực hiện thật sự cho đối tượng Receiver mà đối tượng lệnh tác động lên nó.

Vì yêu cầu được đóng gói, ta có thể lưu trữ, xóa bỏ, làm lại, truy cập, thay đổi, cho áp dụng, ... một yêu cầu.

Do tính chất linh hoạt đó, mẫu thiết kế Command được sử dụng để:

- Gửi yêu cầu đến các đối tượng nhận khác nhau.
- Tạo hàng đợi yêu cầu, ghi nhận (logging) yêu cầu và từ chối yêu cầu.
- Tích hợp giao tác cấp cao từ các tác vụ cơ bản.
- Tạo chuỗi thao tác ghi sẵn (macro recording).
- Cài đặt chức năng Redo và Undo nhiều cấp.

1. Cài đặt



- Command: khai báo giao diện cho việc thực thi một yêu cầu, ví dụ phương thức execute().
- ConcreteCommand: loại Command cụ thể, đã được "tiêm" đối tượng nhận yêu cầu (Receiver) thích hợp. Phương thức execute() của ConcreteCommand được cài đặt bằng cách gọi các phương thức action() của đối tượng Receiver đã "tiêm" vào nó.
- Invoker: đối tượng triệu gọi yêu cầu (Command), đóng gói trong nó các đối tượng Command để triệu gọi. Invoker có thể có một danh sách lưu trữ các Command, ví dụ cho tác vụ undo và redo. Invoker còn gọi là Command Manager.
- Receiver: đối tượng nhận yêu cầu từ Command, dùng các phương thức của mình (action()) để thực hiện yêu cầu đó.
- Client: tạo ra các đối tượng Invoker, rồi dùng Invoker triệu gọi các đối tượng ConcreteCommand mong muốn.

```
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;
```

```
// (1) Invoker
class Invoker {
    private Stack<Command> history = new Stack<>();
    private Stack<Command> redoList = new Stack<>();
    public void invoke(Command command) {
        command.execute();
        if (!(command instanceof UndoCommand) && !(command instanceof RedoCommand))
            history.push(command);
    }

    public void undo() {
        if (!history.isEmpty()) {
            Command command = history.pop();
            command.undo();
            redoList.push(command);
        }
    }

    public void redo() {
        if (!redoList.isEmpty()) {
            Command command = redoList.pop();
            command.execute();
            history.push(command);
        }
    }
}
```

```
// (2) Command
abstract class Command {
    public abstract void execute();
}
```

```

    public abstract void undo();
}

// (3) ConcreteCommand
class UndoCommand extends Command {
    private Invoker receiver;
    public UndoCommand(Invoker receiver) {
        this.receiver = receiver;
    }

    @Override public void execute() { receiver.undo(); }
    @Override public void undo() { }
}

class RedoCommand extends Command {
    private Invoker receiver;
    public RedoCommand(Invoker receiver) {
        this.receiver = receiver;
    }

    @Override public void execute() { receiver.redo(); }
    @Override public void undo() { }
}

class InsertCommand extends Command {
    private Document document;
    String content;
    public InsertCommand(Document document, String content) {
        this.document = document;
        this.content = content;
    }

    @Override public void execute() { document.insert(content); }
    @Override public void undo() { document.remove(); }
}

// (4) Receiver
class Document {
    private List<String> list = new LinkedList<>();
    public void insert(String str) {
        list.add(str);
    }

    public void remove() {
        list.remove(list.size() - 1);
    }

    @Override public String toString() {
        StringBuilder sb = new StringBuilder();
        for (String str : list) sb.append(str).append("\n");
        return sb.toString();
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("---- Command Pattern ----");
        Document doc = new Document();
        Invoker invoker = new Invoker();
        UndoCommand undo = new UndoCommand(invoker);
        RedoCommand redo = new RedoCommand(invoker);
        invoker.invoke(new InsertCommand(doc, "Java Programming Language"));
        invoker.invoke(new InsertCommand(doc, "Object-Oriented Programming"));
        invoker.invoke(new InsertCommand(doc, "Design Patterns"));
        System.out.println("Insert 3 lines:\n" + doc);
        invoker.invoke(undo);
        invoker.invoke(undo);
        System.out.println("Undo 2 times:\n" + doc);
        invoker.invoke(redo);
        System.out.println("Redo 1 time:\n" + doc);
    }
}

```



```
}
```

Mỗi lệnh InsertCommand triệu gọi được lưu vào stack history, Receiver của lệnh này là Document. InsertCommand tác động vào list document của Document.

Các lệnh UndoCommand và RedoCommand không cần lưu vào stack history, Receiver của các lệnh này là Invoker vì chúng tác động vào các stack history và redoList của Invoker. redoList lưu các InsertCommand được undo để redo nếu cần.

2. Liên quan

- Composite: có thể được dùng để cài đặt các vĩ lệnh (macro), kết hợp của nhiều lệnh. Ví dụ: lệnh restart() là kết quả của việc gọi lệnh stop() rồi gọi lệnh start().
- Memento: có thể giữ trạng thái của lệnh, cần cho tác vụ khôi phục (undo).
- Prototype: Command có thể được nhân bản bằng Prototype rồi đặt vào danh sách history.

3. Java API

java.lang Runnable.

Giao diện javax.swing.Action, thừa kế giao diện ActionListener, chính là giao diện Command trong mẫu thiết kế Command. Trong lớp dẫn xuất, cài đặt phương thức actionPerformed(ActionEvent).

```
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
```

```
class FileAction extends AbstractAction {
    FileAction(String name) {
        super(name);
    }

    @Override public void actionPerformed(ActionEvent ae) {
        // add action logic here
    }
}
```

Thêm action vào thanh menu:

```
import java.awt.Event;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;
// ...
JMenu fileMenu = new JMenu("File");
FileAction newAction = new FileAction("New");
JMenuItem item = fileMenu.add(newAction);
item.setAccelerator(KeyStroke.getKeyStroke('N', Event.CTRL_MASK));
// To add action to a toolbar
JToolBar toolbar = new JToolBar();
toolbar.add(newAction);
```

4. Sử dụng

Ta có:

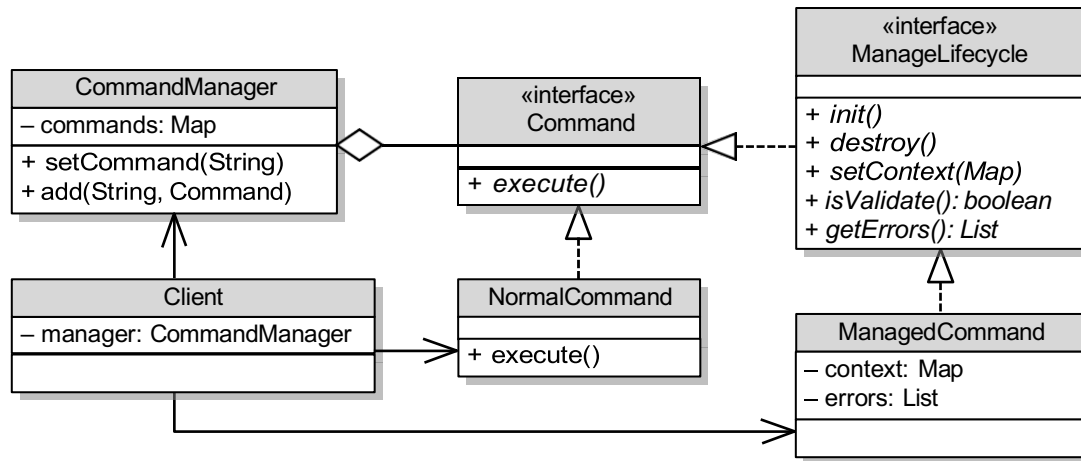
- Các lệnh với đối tượng nhận khác nhau sẽ được xử lý theo cách khác nhau.
- Một tập lệnh cấp cao được thực thi bởi các tác vụ cơ bản.

Ta muốn:

- Chỉ định, xếp thứ tự thực thi và thực thi các lệnh vào những thời điểm khác nhau.
- Hỗ trợ chức năng Undo cho các lệnh.
- Hỗ trợ ghi nhận, kiểm soát các thay đổi do tác động của lệnh.

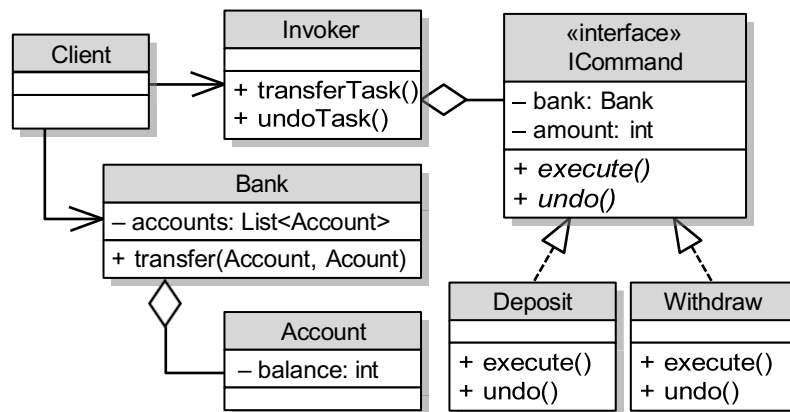
5. Bài tập

a) Mẫu thiết kế Command có thể được mở rộng, hỗ trợ thêm một số dịch vụ cho Command như xác thực quyền, kiểm thử dữ liệu nhập, ghi nhận thực hiện lệnh.



Điều này được thực hiện bằng cách dùng giao diện **ManageLifecycle**, mở rộng giao diện **Command**. Giao diện **ManageLifecycle** cung cấp nhiều phương thức, sẽ được gọi cho mỗi yêu cầu, thay vì chỉ gọi một phương thức **execute()**. Cụ thể, đối với **Command** mở rộng (**Command** được quản lý, **ManagedCommand**), trước khi gọi **execute()** ta có thể xác thực hoặc kiểm thử, sau khi gọi **execute()** ta có thể ghi nhận, thu dọn ngữ cảnh.

b) Các tác vụ nạp tiền (deposit) và rút tiền (withdraw) từ tài khoản (**Account**) của một ngân hàng (**Bank**) có thể đóng gói vào đối tượng (**ICommand**) để tạo nên các tác vụ phức tạp hơn như tác vụ chuyển tiền (transfer) giữa hai tài khoản và hồi tác (undo) việc chuyển tiền. Hãy áp dụng mẫu thiết kế Command để thực hiện yêu cầu này.



Interpreter

Interpret a language using its grammar 

Mẫu thiết kế Interpreter cài đặt bộ dịch cho một ngôn ngữ. Dịch (interpret) được hiểu như là một thao tác trên biểu thức của ngôn ngữ đó, kết quả việc dịch:

- Định trị biểu thức, ví dụ định trị một biểu thức postfix.
- Thay đổi cấu trúc biểu thức, ví dụ chuyển biểu thức infix thành postfix.
- Kiểm tra cú pháp, ví dụ kiểm tra tính hợp lệ một biểu thức.
- Chuyển sang ngôn ngữ khác, ví dụ chuyển thông tin từ một đối tượng thành XML.

Để cài đặt cho bộ dịch, cần định nghĩa ngữ pháp (grammar) của ngôn ngữ nguồn. Ví dụ, ngữ pháp của biểu thức số học infix.

Grammar:

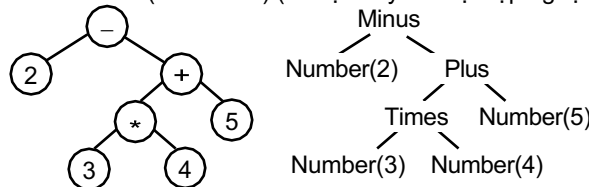
```
Expression ::= Plus | Minus | Times | Div | Number
Plus ::= Expression '+' Expression
Minus ::= Expression '-' Expression
Times ::= Expression '*' Expression
Div ::= Expression '/' Expression
Number ::= int
```

Ngữ pháp của các ngôn ngữ phức tạp thường tạo thành cấu trúc cây:

- Thành phần non-terminal, như node trong của cây, thường định nghĩa đệ quy. Ví dụ, các biểu thức Plus, Minus, Times, Div.
- Thành phần terminal, như node lá của cây. Ví dụ, biểu thức Number.

Mẫu thiết kế Interpreter định nghĩa ngữ pháp bằng cách cài đặt với một hệ thống phân cấp các lớp, mỗi lớp xử lý một thành phần ngữ pháp.

Biểu thức đầu vào của bộ dịch được viết bằng ngôn ngữ nguồn, nói chung được thể hiện thành cây cú pháp trừu tượng (AST – Abstract Syntax Tree) với các thành phần lấy từ ngữ pháp của ngôn ngữ đó. Mỗi biểu thức đầu vào tạo thành cây AST khác nhau. Ví dụ, cây AST của biểu thức đầu vào: $2 - (3 * 4 + 5)$ (khi tạo cây đã loại cặp ngoặc).



Bạn có thể dùng công cụ tại: <http://mshang.ca/syntaxtree/>, nhập vào biểu thức sau để thấy cây AST.

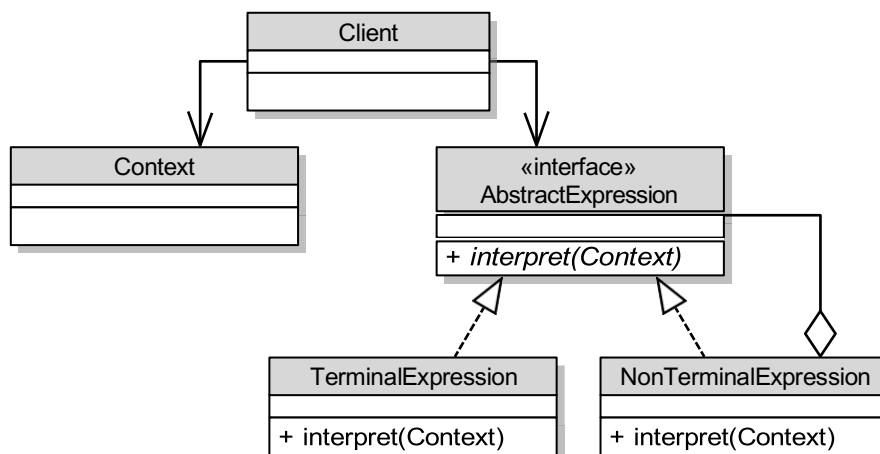
```
[Minus [Number(2)] [Plus [Times [Number(3)] [Number(4)]] [Number(5)]]]
```

GoF không mô tả giải thuật tạo và duyệt cây AST, bạn phải:

- Dùng giải thuật phù hợp. Ví dụ trên, dùng giải thuật Dijkstra tạo cây AST từ biểu thức infix.
- Dùng mẫu thiết kế Visitor.

Khi duyệt AST, mẫu thiết kế Interpreter sẽ chạy tác vụ dịch (interpret), chuyển từng thành phần của ngôn ngữ nguồn thành kết quả đích.

1. Cài đặt



- AbstractExpression: khai báo tác vụ dịch interpret() trừu tượng dùng chung cho tất cả các node của AST.

- TerminalExpression: cài đặt tác vụ interpret() liên kết với các ký hiệu terminal trong ngữ pháp của ngôn ngữ. Một thể hiện của lớp này tương ứng một ký hiệu terminal.

- NonTerminalExpression: cài đặt tác vụ interpret() liên kết với các ký hiệu non-terminal trong ngữ pháp của ngôn ngữ, thường viết đệ quy. Một thể hiện của lớp này tương ứng một luật trong ngữ pháp.

- Context: lưu thông tin toàn cục cho việc dịch, thường chứa chuỗi nhập và kết quả. Lưu ý là không nhất thiết phải có lớp Context, bạn có thể truyền chuỗi nhập như tham số và lấy kết quả như trị trả về.

Trong ví dụ dưới, ngữ pháp là đơn giản, chỉ có các lớp TerminalExpression.

Grammar:

```
DateExpression ::= WordDate | CalendarDate | GregorianDate
WordDate ::= String // 124th day, Thursday, May 4, 2017
CalendarDate ::= String // 5-4-2017
GregorianDate ::= String // 5-4-2017 12:40:5
```

Phương thức dịch (interpret) của các lớp TerminalExpression sẽ "dịch" đối tượng Date đầu vào thành các chuỗi có định dạng thích hợp.

```
import java.util.Date;
import java.util.Calendar;

// (1) Context
class Context {
    String formattedDate;
    Date date;

    public Context(Date date) {
        this.date = date;
    }
}

// (2) AbstractExpression
abstract class AbstractDateExpression {
    static final String[] convertDayOfWeek = {
        "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
    static final String[] convertMonth = {
        "January", "February", "March", "April", "May", "June", "July",
        "August", "September", "October", "November", "December" };
    public abstract void interpret(Context context);
}

// (3) TerminalExpression
class WordDateExpression extends AbstractDateExpression {
    @Override public void interpret(Context context) {
        Calendar c = Calendar.getInstance();
        c.setTime(context.date);
        context.formattedDate = c.get(Calendar.DAY_OF_YEAR) + "th day, " +
            convertDayOfWeek[c.get(Calendar.DAY_OF_WEEK) - 2] + ", " +
            convertMonth[c.get(Calendar.MONTH)] + " " +
            c.get(Calendar.DATE) + ", " + c.get(Calendar.YEAR);
    }
}

class CalendarDateExpression extends AbstractDateExpression {
    @Override public void interpret(Context context) {
        Calendar c = Calendar.getInstance();
        c.setTime(context.date);
        context.formattedDate =
            c.get(Calendar.MONTH) + 1 + "-" + c.get(Calendar.DATE) + "-" + c.get(Calendar.YEAR);
    }
}

class GregorianCalendarExpression extends AbstractDateExpression {
    @Override public void interpret(Context context) {
        Calendar c = Calendar.getInstance();
        c.setTime(context.date);
        context.formattedDate =
            c.get(Calendar.MONTH) + 1 + "-" + c.get(Calendar.DATE) + "-" +
            c.get(Calendar.YEAR) + " " + c.get(Calendar.HOUR_OF_DAY) + ":" +
            c.get(Calendar.MINUTE) + ":" + c.get(Calendar.SECOND);
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Interpreter Pattern ---");
        Context context = new Context(new Date());
        new WordDateExpression().interpret(context);
        System.out.println("Word Date: " + context.formattedDate);
        new CalendarDateExpression().interpret(context);
        System.out.println("Calendar Date: " + context.formattedDate);
        new GregorianCalendarExpression().interpret(context);
        System.out.println("Gregorian Date: " + context.formattedDate);
    }
}
```

2. Liên quan

- Composite: cây AST được xây dựng bằng mẫu thiết kế Composite.
- Flyweight: áp dụng để tạo phần dùng chung cho các ký hiệu terminal trong cây AST.
- Iterator: dùng để duyệt cây AST.
- Visitor: dùng đóng gói hành vi xử lý từng node trên cây AST.

3. Java API

java.util.regex.Pattern và java.util.regex.Matcher dùng một thiết kế Interpreter nội.

java.text.Normalizer.

Tất cả lớp con của java.text.Format.

4. Sử dụng

Ta có:

- Ngữ pháp của ngôn ngữ cần diễn dịch không quá phức tạp.
- Hiệu quả diễn dịch, như tốc độ dịch, không phải là yêu cầu chính.

5. Bài tập

a) Biểu thức đầu vào là một biểu thức số học infix, có ngữ pháp như sau:

Grammar:

```
Expression ::= Plus | Minus | Times | Div | Number
Plus ::= Expression '+' Expression
Minus ::= Expression '-' Expression
Times ::= Expression '*' Expression
Div ::= Expression '/' Expression
Number ::= int
```

Áp dụng mẫu thiết kế Interpreter, viết chương trình thực hiện các tác vụ "dịch" sau:

- evaluation(), "dịch" biểu thức infix sang trị.
- preorder(), "dịch" biểu thức infix sang biểu thức prefix.
- postorder(), "dịch" biểu thức infix sang biểu thức postfix.

Chi tiết về các thuật toán "dịch" trình bày trong tài liệu Cấu trúc dữ liệu và thuật toán, cùng người viết.

b) Ngôn ngữ nguồn là số La mã, có ngữ pháp như sau (λ là rỗng):

Grammar:

```
Thousand ::= 'M' Thousand | λ; // 1000 →
Hundred ::= 'C'D' | 'C'M' | 'D' le300 | le300; // 400 | 900 | 500 - 800 | 000 - 300
le300 ::= λ | 'C'C'C' | 'C'C' | 'C';
Ten ::= le30 | 'X'L' | 'L' le30 | 'X'C'; // 00 - 30 | 40 | 50 - 80 | 90
le30 ::= λ | 'X'X'X' | 'X'X' | 'X';
Unit ::= le3 | 'I'V' | 'V' le3 | 'I'X'; // 0 - 3 | 4 | 5 - 8 | 9
le3 ::= λ | 'I'I'I' | 'I'I' | 'I';
```

Áp dụng mẫu thiết kế Interpreter, viết chương trình dịch số La mã sang số thập phân (Arabic) hiện dùng.

c) Cho ngữ pháp của biểu thức Boolean.

Grammar:

```
e ::= e '&' e
    | e '|' e
    | '!' e
    | '(' e ')'
    | var '=' e
    | var
```

Áp dụng mẫu thiết kế Interpreter, cài đặt tác vụ "dịch" là định trị một biểu thức Boolean.

Biểu thức Boolean, ví dụ: X = (A & B) | !C được tạo như sau:

```
Logic term = new ANDLogic(new Variable("A"), new Variable("B"));
term = new ORLogic(term, new NOTLogic( new Variable("C")));
term = new AssignmentLogic(new Variable("X"), term);
```

Iterator

Access aggregated objects 

Mẫu thiết kế Iterator cung cấp một cách truy cập thống nhất đến các đối tượng thành phần, nằm bên trong một đối tượng chứa (container hoặc collection), mà không cần phải hiểu rõ đến cấu trúc nội tại của đối tượng chứa đó.

Mặc dù một đối tượng chứa thường có đủ phương thức để truy cập các đối tượng thành phần của nó, ví dụ:

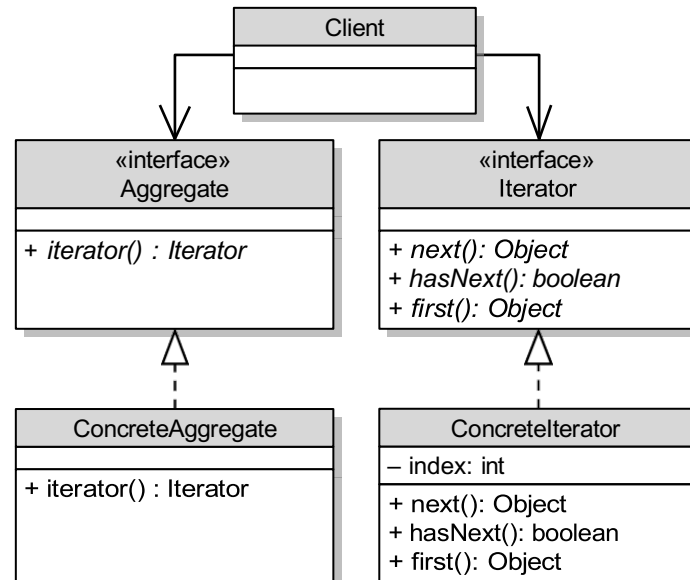
```
ArrayList<Book> books = Bookstore.getBooks();
```

```
for (int i = 0; i < books.size(); ++i)
```

```
System.out.println(i + ": " + books.get(i));
```

nhưng để bảo đảm nguyên tắc SRP, tác vụ duyệt các đối tượng thành phần của đối tượng chứa được tách ra và đóng gói vào một đối tượng gọi là Iterator. Iterator được hình dung như một "con trỏ" dịch chuyển trong đối tượng chứa, dùng để truy cập các đối tượng thành phần của đối tượng chứa. Do đặc điểm này, Iterator còn gọi là Cursor.

1. Cài đặt



- Iterator: định nghĩa một giao diện chuẩn để truy cập và duyệt các đối tượng thành phần của đối tượng chứa. Các tác vụ điển hình bao gồm: trở đến đối tượng thành phần kế tiếp (`next()`), kiểm tra xem có đối tượng thành phần kế tiếp không (`hasNext()`).

- ConcreteIterator: cài đặt cho giao diện Iterator, giữ tham chiếu chỉ đến vị trí hiện tại khi duyệt đối tượng chứa, tức vị trí của đối tượng thành phần mà Iterator hiện đang trở đến.

- Aggregate: giao diện của đối tượng chứa, khai báo phương thức `iterator()`, trả về Iterator, là đối tượng dùng duyệt các đối tượng thành phần của nó.

- ConcreteAggregate: cài đặt giao diện Aggregate để tạo đối tượng Iterator, trả về một đối tượng ConcreteIterator cụ thể. Khi trả về Iterator, Aggregate trao cho Iterator này tham chiếu chỉ đến chính nó để Iterator có thể duyệt các đối tượng thành phần của Aggregate đó.

Khi sử dụng, Client gọi phương thức `iterator()` của đối tượng chứa Aggregate để nhận được đối tượng Iterator dùng duyệt nó. Iterator này được gọi là external iterator (iterator chủ động), được điều khiển bởi Client. Ngoài ra, còn các internal iterator (iterator thụ động) được điều khiển bởi Aggregate.

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
```

```
// (1) Aggregate
interface CollectionIF {
    IteratorIF iterator();
    Collection elements();
}
```

```
// (2) Iterator
interface IteratorIF {
    boolean hasNext();
    Object next();
}
```

```
// (3) ConcreteAggregate
class ConcreteCollection implements CollectionIF {
    private List list;
    public ConcreteCollection(Object[] objectList) {
        list = Arrays.asList(objectList);
    }
}
```

```

    @Override public IteratorIF iterator() {
        return new ConcreteIterator(this);
    }

    @Override public Collection elements() {
        return Collections.unmodifiableList(list);
    }
}

// (4) ConcreteIterator
class ConcreteIterator implements IteratorIF {
    private List list;
    private int index;
    public ConcreteIterator(CollectionIF collection) {
        list = (List)collection.elements();
        index = 0;
    }

    @Override public boolean hasNext() {
        return (index < list.size());
    }

    @Override public Object next() {
        try {
            return list.get(index++);
        } catch (IndexOutOfBoundsException e) {
            throw new RuntimeException("No Such Element");
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Iterator Pattern ---");
        String[] books = { "Sequential", "Procedural", "OOP", "Design Patterns" };
        CollectionIF collection = new ConcreteCollection(books);
        System.out.println("Getting an iterator for the collection...");
        IteratorIF iterator = collection.iterator();
        System.out.println("Iterate through the list.");
        for (int i = 0; iterator.hasNext(); ++i)
            System.out.println(i + ": " + iterator.next());
    }
}

```

2. Liên quan

- Composite: Iterator thường dùng để duyệt một cấu trúc đệ quy như các Composite.
- Factory Method: phương thức iterator() là một Factory Method của Aggregate, lớp dẫn xuất của nó mới quyết định việc tạo Iterator thích hợp cho loại ConcreteAggregate tương ứng.
- Memento: thường dùng cùng với Iterator. Iterator có thể sử dụng Memento để lưu trạng thái duyệt.

3. Java API

Mẫu thiết kế Iterator đã được tích hợp vào Java API, các lớp Collection đều có phương thức iterator() trả về thực thể cài đặt giao diện java.util.Iterator dùng truy cập các đối tượng thành phần của Collection đó.

Hơn thế nữa, nếu người dùng tạo các collection tùy biến riêng, Java API hỗ trợ người dùng áp dụng mẫu thiết kế Iterator, dễ dàng tạo iterator cho collection, cho phép dùng cả vòng lặp for tăng cường (foreach) tiện dụng.

Để áp dụng mẫu thiết kế Iterator:

- Collection của bạn phải cài đặt giao diện java.lang.Iterable, bạn cần hiện thực phương thức iterator() trả về một java.util.Iterator. Giao diện Iterable chính là giao diện Aggregate của mẫu thiết kế Iterator.
- ConcreteIterator phải cài đặt giao diện java.util.Iterator, cần hiện thực các phương thức hasNext(), next() và remove(). Giao diện Iterator chính là giao diện Iterator của mẫu thiết kế Iterator.

Ví dụ sau đây, viết lại ví dụ trên nhưng áp dụng mẫu thiết kế Iterator được tích hợp trong Java API.

```

import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

interface CollectionIF<E> extends Iterable<E> {
    Collection<E> elements();
}

```

```

class ConcreteCollection<E> implements CollectionIF<E> {
    private List<E> list;
    public ConcreteCollection(E[] objectList) {
        list = Arrays.asList(objectList);
    }

    @Override public Iterator<E> iterator() {
        return new ConcreteIterator(this);
    }

    @Override public Collection<E> elements() {
        return Collections.unmodifiableList(list);
    }
}

class ConcreteIterator<E> implements Iterator<E> {
    private List<E> list;
    private int index;
    public ConcreteIterator(CollectionIF<E> collection) {
        list = (List<E>)collection.elements();
        index = 0;
    }

    @Override public boolean hasNext() {
        return (index < list.size());
    }

    @Override public E next() {
        try {
            return list.get(index++);
        } catch (IndexOutOfBoundsException e) {
            throw new RuntimeException("No such element");
        }
    }

    @Override public void remove() { }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Iterator Pattern in Java API ---");
        String[] books = {"Sequential", "Procedural", "OOP", "Design Patterns"};
        CollectionIF<String> collection = new ConcreteCollection(books);
        System.out.println("Foreach through the list.");
        int i = 0;
        for (String s : collection) System.out.println((i++) + ": " + s);
    }
}

```

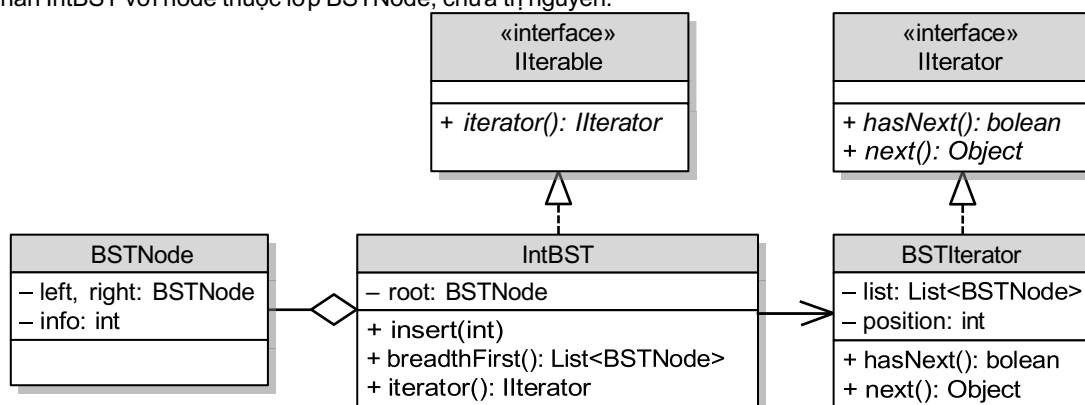
4. Sử dụng

Ta muốn:

- Truy cập các đối tượng thành phần của đối tượng chứa mà không bộc lộ cấu trúc bên trong đối tượng chứa.
- Hỗ trợ nhiều phương án duyệt của các đối tượng thành phần của một đối tượng chứa.
- Cung cấp một giao diện đơn giản, tổng quát cho nhiều kiểu đối tượng chứa có cấu trúc khác nhau.

5. Bài tập

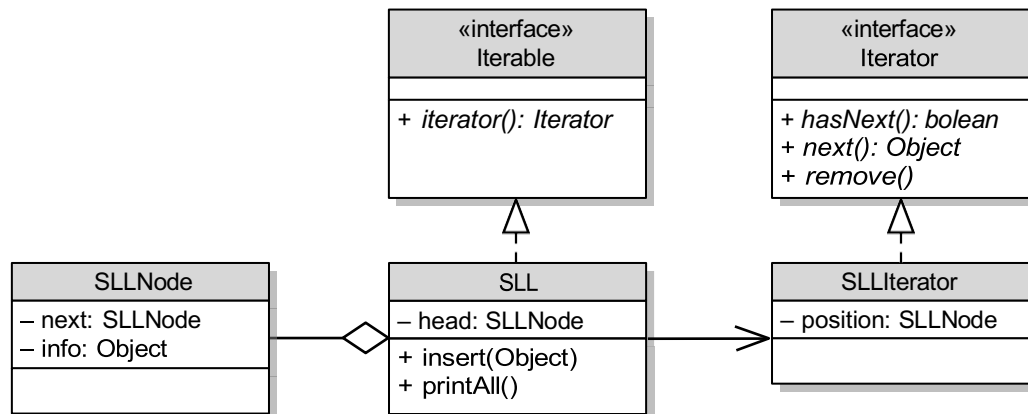
a) Tạo nhị phân IntBST với node thuộc lớp BSTNode, chứa trị nguyên.



Áp dụng mẫu thiết kế Iterator để có thể lấy được đối tượng BSTIterator từ cây IntBST. Đối tượng này cho phép duyệt các node của cây InBST theo mức.

b) Tạo một danh sách liên kết đơn SLL (Singly Linked List) với các phần tử là các node thuộc lớp SLLNode (xem tài liệu "Cấu trúc dữ liệu và thuật toán", cùng người viết).

Áp dụng framework Iterator của Java API để tạo một Iterator cho SLL, cho phép duyệt SLL bằng vòng lặp for tăng cường.



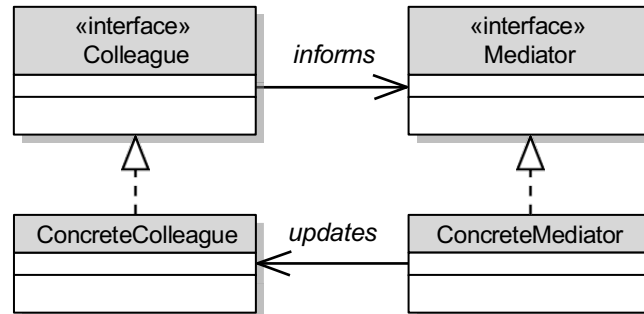
Mediator

Define object interaction 

Mẫu thiết kế Mediator dùng giải quyết độ phức tạp trong tương tác, liên lạc trực tiếp giữa các đối tượng/lớp. Mẫu thiết kế này cung cấp một lớp giữ vai trò trung gian (Mediator) giữa các đối tượng/lớp, đóng gói cách mà một tập đối tượng tương tác với nhau, bao gồm thông tin trao đổi, hành vi liên lạc giữa các đối tượng/lớp.

Mẫu thiết kế Mediator giúp đơn giản hóa giao thức liên lạc giữa các đối tượng/lớp, điều khiển tập trung tương tác giữa chúng, loại bỏ các thao tác liên lạc đặc thù của các đối tượng/lớp.

1. Cài đặt



- Mediator: khai báo giao diện cho việc liên lạc giữa các đối tượng Colleague.
- ConcreteMediator: cài đặt các hành vi liên lạc giữa các Colleague và tham chiếu đến các Colleague có liên quan.
- Colleague: mỗi Colleague biết đối tượng Mediator của nó và nó giao tiếp với đối tượng Mediator này khi muốn liên lạc với đối tượng Colleague khác.

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
// (1) Mediator
interface Mediator<T> {
    void sendMessage(User<T> user, T message);
    void addUser(User<T> user);
}

// (2) ConcreteMediator
class ChatMediator<T> implements Mediator<T> {
    private List<User<T>> userList = new ArrayList<>();
    @Override public void addUser(User<T> user) {
        userList.add(user);
    }

    @Override public void sendMessage(User<T> user, T message) {
        for (User<T> u : userList)
            if (u != user) u.receive(message);
    }
}

// (3) Colleague
abstract class User<T> {
    protected String name;
    public User(String name) {
        this.name = name;
    }

    public abstract void send(Mediator<T> mediator, T message);
    public abstract void receive(T message);
}

// (4) ConcreteColleague
class ChatUser<T> extends User<T> {
    public ChatUser(String name) {
        super(name);
    }

    @Override public void send(Mediator<T> mediator, T message) {
        mediator.sendMessage(this, message);
    }

    @Override public void receive(T message) {
        System.out.println(this.name + " received: " + message);
    }
}
```

```

    }
}

class Message {
    private String message;
    public Message(String message) {
        this.message = new Date() + ", " + message;
    }

    @Override public String toString() { return message; }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Mediator Pattern ---");
        ChatUser<Message> obama = new ChatUser<>("Barack Obama");
        ChatUser<Message> un = new ChatUser<>("Kim Jong Un");
        ChatUser<Message> putin = new ChatUser<>("Vladimir Putin");
        ChatMediator<Message> msn = new ChatMediator<>();
        msn.addUser(obama);
        msn.addUser(putin);
        msn.addUser(un);
        ChatMediator<Message> yahoo = new ChatMediator<>();
        yahoo.addUser(putin);
        yahoo.addUser(un);
        un.send(msn, new Message("[Kim Jong Un]: Ultimate Letter"));
        un.send(yahoo, new Message("[Kim Jong Un]: Secret Letter"));
    }
}

```

2. Liên quan

- Facade: mẫu thiết kế Facade trừu tượng hóa một hệ thống con để cung cấp một giao diện dễ dùng hơn, đây là giao thức một hướng (Client → Facade → Subsystem). Khác với Facade, Mediator là trung gian giao tiếp giữa các Colleague, đây là giao thức đa chiều.
- Observer: các Colleague có thể liên lạc với nhau bằng cách dùng mẫu thiết kế Observer.

3. Java API

java.util.Timer, các phương thức scheduleXxx().
 java.lang.reflect.Method, phương thức invoke().

4. Sử dụng

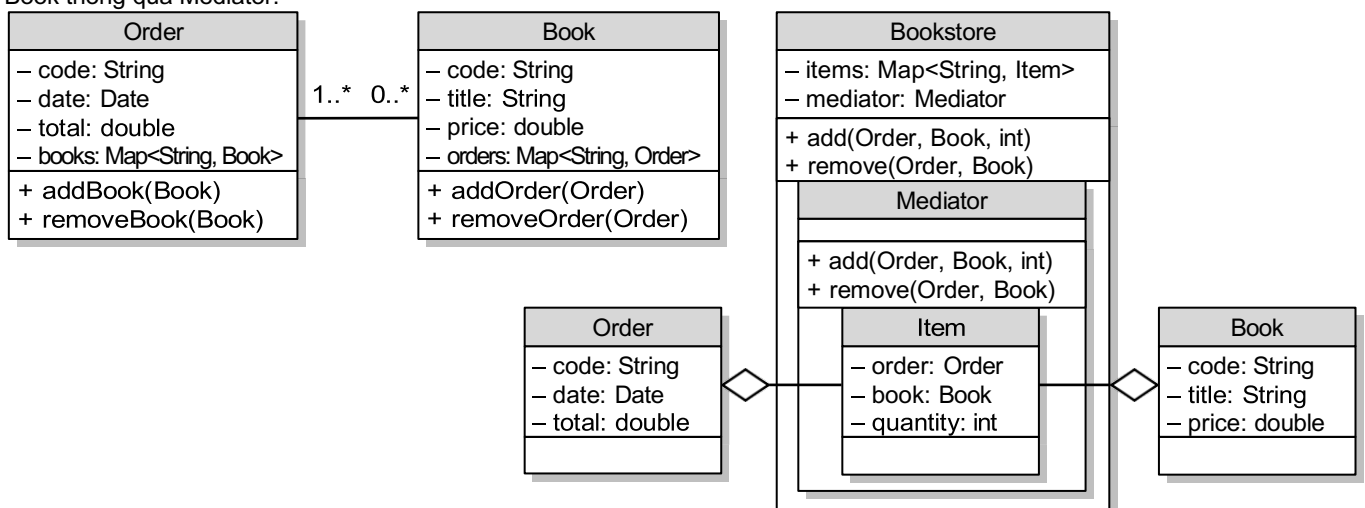
Ta có:

- Một tập các đối tượng liên lạc với nhau theo những cách có cấu trúc tốt nhưng lại phức tạp.
- Cần phải tùy biến hành vi liên lạc của nhóm đối tượng mà không phải dẫn xuất chúng.
- Một hệ thống hoạt động dựa trên thông điệp.

5. Bài tập

Sơ đồ bên trái thể hiện quan hệ nhiều-nhiều giữa hai lớp Order và Book, làm cả hai phải giữ một Map lưu trữ thông tin của nhau. Tương tác trực tiếp giữa chúng trở nên phức tạp.

Sơ đồ bên phải áp dụng mẫu thiết kế Mediator để giảm sự phức tạp này. Lớp Bookstore làm việc với quan hệ giữa Order và Book thông qua Mediator.



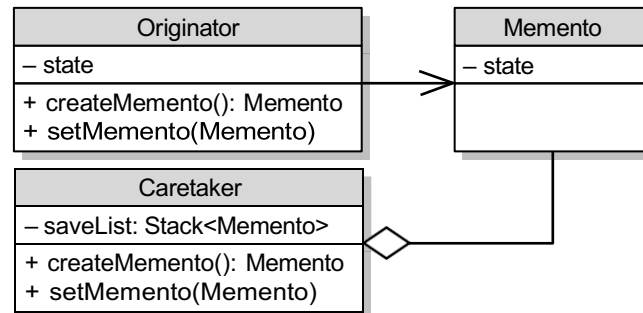
Memento

Externalize object internal state 

Mẫu thiết kế Memento được dùng khi muốn khôi phục lại trạng thái lần trước của một đối tượng. Mẫu thiết kế Memento không vi phạm tính đóng gói (encapsulation) mà vẫn có thể *lấy và lưu trữ trạng thái nội* của một đối tượng, vì vậy có thể khôi phục lại trạng thái đó nếu cần.

Mẫu thiết kế Memento còn gọi là Token.

1. Cài đặt



- Memento: đối tượng Memento chứa các trạng thái cần lưu trữ của một đối tượng (Originator), mỗi Memento thể hiện các trạng thái của đối tượng tại một thời điểm, gọi là "bản chụp" (snapshot) của đối tượng.

Lý tưởng là đảm bảo chỉ Originator mới có quyền truy cập Memento.

- Originator: chính là đối tượng mà ta quan tâm lưu trữ trạng thái. Thường có hai phương thức quan trọng, giúp nó lưu trữ trạng thái mà không vi phạm tính đóng gói.

+ Tạo đối tượng Memento và gán trạng thái cần lưu trữ vào đối tượng Memento đó. Nói cách khác, ta lưu trạng thái nội của Originator vào đối tượng Memento.

+ Khôi phục trạng thái từ đối tượng Memento được lưu trữ. Nói cách khác, ta khôi phục trạng thái nội trước đây từ thông tin lấy từ đối tượng Memento.

- Caretaker: giữ một ArrayList hoặc Stack lưu giữ các phiên bản trước của Memento. Dùng nó để lưu trữ và tìm lại các Memento đã lưu. Caretaker không quan tâm đến trạng thái được lưu trữ, mà chỉ quan tâm đến các Memento được nó lưu trữ.

Caretaker yêu cầu một hành động lưu trữ, yêu cầu một "bản chụp" của Originator tại một thời điểm.

// (1) Memento

```

class Memento {
    private String name;
    private double cost;
    public Memento(Product product) {
        this.name = product.getName();
        this.cost = product.getCost();
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getCost() { return cost; }
    public void setCost(double cost) { this.cost = cost; }
}
  
```

// (2) Originator

```

class Product {
    private String name;
    private double cost;
    public Product(String name, double cost) {
        this.name = name;
        this.cost = cost;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getCost() { return cost; }
    public void setCost(double cost) { this.cost = cost; }

    @Override
    public String toString() {
        return String.format("%s [%s]", name, cost);
    }

    public Memento createMemento() {
        return new Memento(this);
    }

    public void setMemento(Memento memento) {
  
```

```

    this.setName(memento.getName());
    this.setCost(memento.getCost());
}
}

// (3) Caretaker
class Caretaker {
    private java.util.Stack<Memento> saveList = new java.util.Stack<>();
    public void add(Memento memento) {
        saveList.push(memento);
    }

    public Memento get() {
        return saveList.isEmpty() ? null : saveList.pop();
    }
}

public class Client {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();
        Product product = new Product("Book", 50.00);
        System.out.println(product);
        System.out.println("Change the object: ");
        caretaker.add(product.createMemento());
        product.setName("Meat");
        caretaker.add(product.createMemento());
        product.setCost(60.00);
        System.out.println(product);
        System.out.println("Restore state via the memento...");
        product.setMemento(caretaker.get());
        product.setMemento(caretaker.get());
        System.out.println(product);
    }
}

```

2. Liên quan

- Command: dùng Memento để lưu trạng thái các tác vụ có thể khôi phục lại (undo).
- Iterator: Memento có thể được dùng cho thao tác lặp.

3. Java API

Các lớp cài đặt giao diện java.io.Serializable.

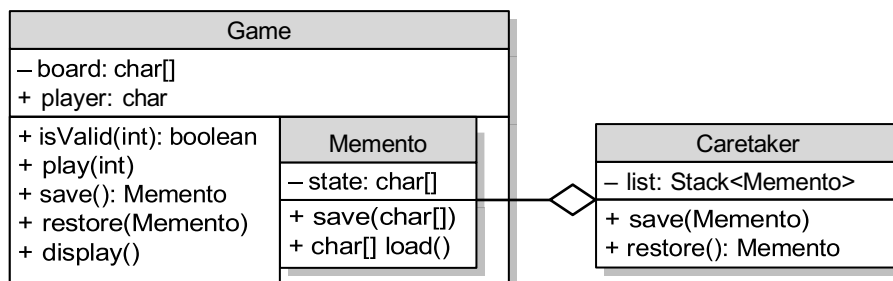
4. Sử dụng

Ta muốn:

- Lưu trữ bản sao trạng thái của một đối tượng để có thể khôi phục lại sau này (chức năng Undo).
- Thay đổi, khôi phục trạng thái của một đối tượng mà không can thiệp trực tiếp đến trạng thái đó.

5. Bài tập

Trò chơi TicTacToe áp dụng mẫu thiết kế Memento để lưu giữ trạng thái bàn cờ (mảng board) và người chơi (board[0]).



Đến lượt chơi của mình, người chơi ('X' hoặc 'O', 'X' đi trước) có các lựa chọn sau:

- Chọn nước đi, nước đi hợp lệ là từ 1 đến 9 và ô chọn còn trống.
- Chọn U để hoàn tác (undo), trạng thái bàn cờ sẽ lui lại 2 bước. Nếu 'O' mới đi một nước, bàn cờ sẽ trở lại trạng thái đầu.
- Chọn Q để thoát trò chơi. Các tùy chọn không hợp lệ sẽ yêu cầu nhập lại

```

--- TicTacToe ---
1 | 2 | 3
-----
4 | 5 | 6
-----
7 | 8 | 9

Player 'X' move? 5
1 | 2 | 3
-----
4 | X | 6
-----
7 | 8 | 9

Player 'O' move? 3
1 | 2 | O
-----
4 | X | 6
-----
7 | 8 | 9

Player 'X' move? U
1 | 2 | 3
-----
4 | 5 | 6
-----
7 | 8 | 9

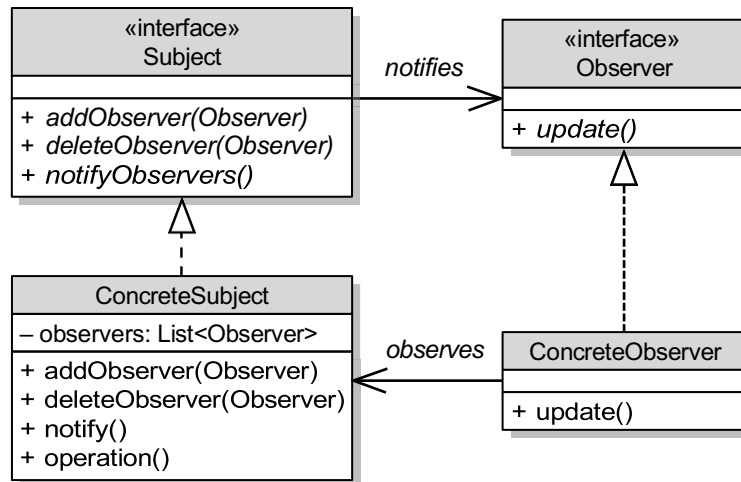
```

Observer

Subscribe to object changes 

Mẫu thiết kế Observer định nghĩa một phụ thuộc một-nhiều, trong đó nếu *một* đối tượng (Subject, còn gọi là Observable) thay đổi trạng thái, tất cả (*nhiều*) các đối tượng (Observer) phụ thuộc đối tượng đó sẽ được thông báo và tự động cập nhật. Phía "một" thường là dữ liệu, phía "nhiều" thường là giao diện người dùng (bảng biểu, đồ thị, báo cáo). Mẫu thiết kế này còn gọi là Dependents, Publisher/Subscriber hoặc Source/Listener.

1. Cài đặt



- Subject: giao diện cho đối tượng dữ liệu, khai báo các phương thức chính:
 - + addObserver(): dùng thêm các Observer vào danh sách đăng ký các đối tượng cần phải thông báo về những thay đổi.
 - + deleteObserver(): loại Observer chỉ định ra khỏi danh sách đăng ký các đối tượng cần thông báo về những thay đổi. Do Observer chứa một tham chiếu đến Subject mà nó đăng ký, nên khi nó không còn quan tâm đến sự thay đổi của Subject nữa, Observer sẽ thông qua tham chiếu đó, tự loại nó ra khỏi danh sách đăng ký với Subject.
 - + notifyObservers(): thông báo cho các Observer trong danh sách đăng ký về những thay đổi trên Subject.
- ConcreteSubject: cài đặt giao diện Subject. Vì thường là đối tượng dữ liệu, nó lưu trữ trạng thái mà các đối tượng Observer quan tâm. Khi trạng thái này thay đổi, các Observer đăng ký với nó, chứa trong một danh sách nó lưu giữ, sẽ được thông báo. Chú ý là danh sách cần thông báo do ConcreteSubject lưu giữ chứa các thực thể kiểu interface Observer, vì vậy cho phép đăng ký các đối tượng của nhiều lớp ConcreteObserver khác nhau, cài đặt phương thức đa hình update() theo cách khác nhau.
- Observer: khai báo giao diện với phương thức chính update(). Phương thức này có thể truy cập đối tượng Subject mà nó đăng ký, cập nhật Observer với trạng thái thay đổi của Subject.
- ConcreteObserver: cài đặt giao diện của Observer. Trong constructor, nó tự đăng ký với đối tượng Subject mà nó theo quan tâm, bằng cách gọi phương thức addObserver() của tham chiếu đến Subject. Khi được thông báo, nó sẽ thực thi một tác vụ nào đó, ví dụ thay đổi giao diện, cập nhật biểu đồ.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
```

```
// (1) Observer
interface Observer {
    void update();
}

// (2) Subject
interface Subject {
    void addObserver(Observer observer);
    void deleteObserver(Observer observer);
    void notifyObservers();
}

// (3) ConcreteObserver
class ConcreteObserver implements Observer {
    private ConcreteSubject subject;
    public ConcreteObserver(ConcreteSubject subject) {
        this.subject = subject;
        this.subject.addObserver(ConcreteObserver.this);
    }

    @Override public void update() {
        System.out.printf("[%0.2f]", subject.d);
    }
}
```

// (4) ConcreteSubject

```

class ConcreteSubject implements Subject {
    double d;
    List<Observer> observers = new ArrayList<>();
    @Override public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override public void deleteObserver(Observer observer) {
        observers.remove(observers.indexOf(observer));
    }

    @Override public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

    public void operation() {
        Random random = new Random();
        d = random.nextDouble();
        if (d < 0.25 || d > 0.75) {
            System.out.print("Yes");
            notifyObservers();
        } else {
            System.out.print("No");
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Observer Pattern ---");
        ConcreteSubject subject = new ConcreteSubject();
        Observer observer1 = new ConcreteObserver(subject);
        Observer observer2 = new ConcreteObserver(subject);
        System.out.println("Doing something in the subject over time...");
        System.out.println("Observable Observer1 Observer2");
        System.out.println("Iteration changed? notified? notified?");
        for (int i = 0; i < 10; ++i) {
            System.out.print(i + ": ");
            subject.operation();
            System.out.println();
        }
        System.out.println("Removing observer1 from the subject... Repeating...");
        System.out.println("Observable Observer2");
        System.out.println("Iteration changed? notified?");
        subject.deleteObserver(observer1);
        for (int i = 0; i < 10; ++i) {
            System.out.print(i + ": ");
            subject.operation();
            System.out.println();
        }
    }
}

```

2. Liên quan

- Mediator: bằng cách đóng gói những cập nhật ngữ cảnh phức tạp, Observable hoạt động như đối tượng Mediator giữa các đối tượng và các Observer.
- Singleton: các Observable có thể là Singleton để nó trở nên duy nhất và được truy cập toàn cục.

3. Java API

Mẫu thiết kế Observer đã được tích hợp vào Java API, gói java.util. Để áp dụng mẫu thiết kế này:

- ConcreteSubject, đối tượng dữ liệu, cần thừa kế lớp Observable. Trong phương thức operation(), sau khi thay đổi dữ liệu, gọi các phương thức setChanged() và notifyObservers() của giao diện Observable để tự động cập nhật cho các đối tượng Observer có đăng ký nhận cảnh báo thay đổi với nó.
- ConcreteObserver, phần hiển thị của ứng dụng (giao diện người dùng GUI, report, sơ đồ, bảng biểu) thường cài đặt giao diện Observer. Trong constructor, nó tự "đăng ký" để nhận cảnh báo thay đổi diễn ra trên đối tượng Observable mà nó quan tâm. Đồng thời, cài đặt phương thức update(Observable, Object), trong đó nó nhận dữ liệu thay đổi từ đối tượng Observable để cập nhật phần hiển thị của mình.

Ví dụ sau đây, viết lại ví dụ trên nhưng áp dụng mẫu thiết kế Observer được tích hợp trong Java API.

```

import java.util.Observable;
import java.util.Observer;
import java.util.Random;

class ConcreteObserver implements Observer {
    public ConcreteObserver(Observable observable) {
        observable.addObserver(ConcreteObserver.this);
    }

    @Override public void update(Observable o, Object arg) {
        if (o instanceof ConcreteSubject) {
            ConcreteSubject t = (ConcreteSubject)o;
            System.out.printf("          [%.2f]", t.d);
        }
    }
}

class ConcreteSubject extends Observable {
    double d;
    public void operation() {
        Random random = new Random();
        d = random.nextDouble();
        if (d < 0.25 || d > 0.75) {
            System.out.print("Yes");
            this.setChanged();
            this.notifyObservers();
        } else {
            System.out.print("No");
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Observer Pattern in Java API ---");
        System.out.println("Doing something in the subject over time...");
        System.out.println("          Observable Observer1 Observer2");
        System.out.println("Iteration changed? notified? notified?");
        ConcreteSubject subject = new ConcreteSubject();
        Observer observer1 = new ConcreteObserver(subject);
        Observer observer2 = new ConcreteObserver(subject);
        for (int i = 0; i < 10; ++i) {
            System.out.print( i + ":      ");
            subject.operation();
            System.out.println();
        }
        System.out.println("Removing observer1 from the subject... Repeating...");
        System.out.println("          Observable Observer2");
        System.out.println("Iteration changed? notified?");
        subject.deleteObserver(observer1);
        for (int i = 0; i < 10; ++i) {
            System.out.print( i + ":      ");
            subject.operation();
            System.out.println();
        }
    }
}

```

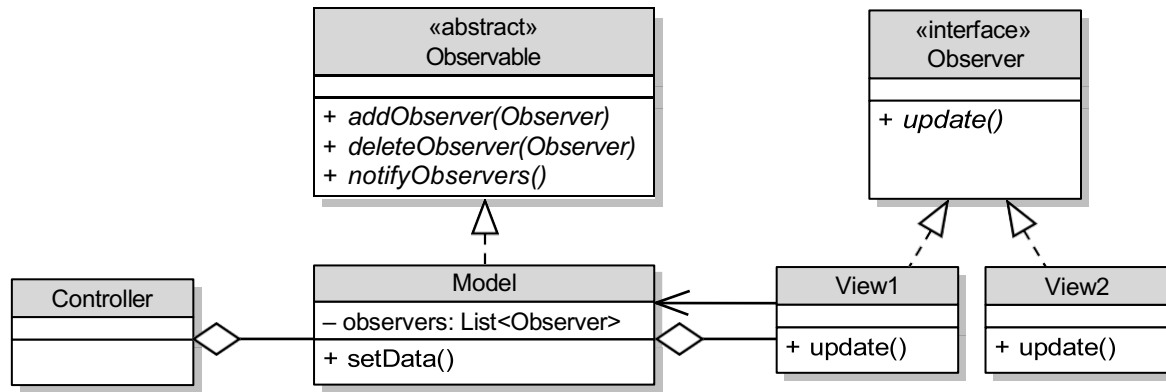
4. Sử dụng

Ta muốn:


- Cập nhật trên một đối tượng (thường là dữ liệu) sẽ thay đổi một số đối tượng được lựa chọn khác (thường là giao diện người dùng), không xác định được số đối tượng thay đổi kéo theo.
- Một đối tượng cần thông báo cho một số các đối tượng khác mà không cần biết thông tin về các đối tượng được thông báo.

5. Bài tập

Sơ đồ sau trình bày mối quan hệ khi phối hợp giữa mẫu thiết kế Observer và mẫu thiết kế MVC. Hãy viết một chương trình dùng Java Swing, truy xuất cơ sở dữ liệu và hiển thị dữ liệu lên bảng và form trong GUI. Áp dụng framework Observer của Java API và mẫu thiết kế MVC như sơ đồ sau.



State

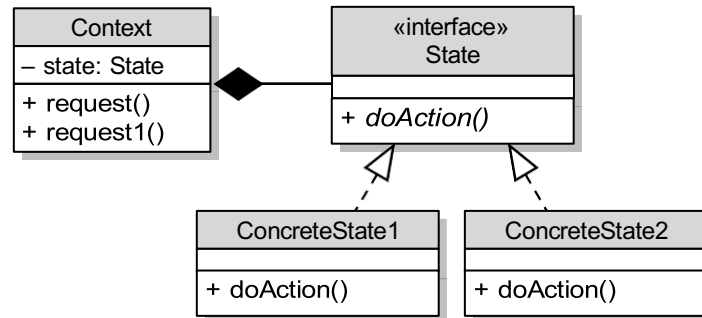
Change behavior based on state 

Đối tượng có trạng thái (state) và hành vi (behavior). Đối tượng thay đổi trạng thái dựa trên các sự kiện bên trong và bên ngoài. Nếu một đối tượng mà sự thay đổi qua lại giữa các trạng thái đã được xác định rõ ràng, và hành vi của đối tượng phụ thuộc đặc biệt vào trạng thái của nó, đối tượng đó là một ứng viên tốt cho mẫu thiết kế State.

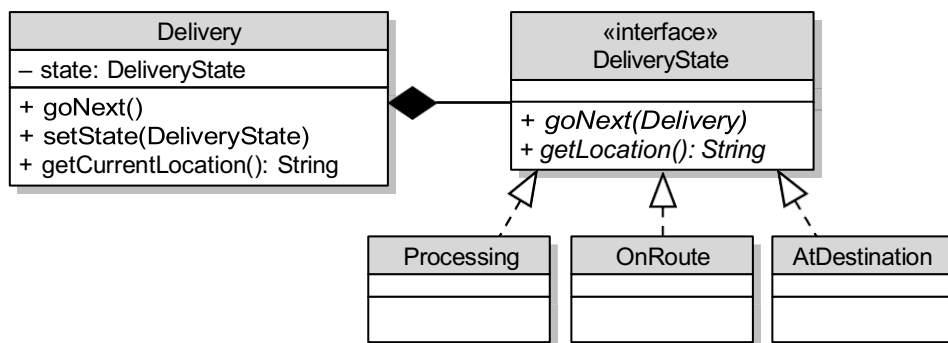
Mẫu thiết kế State tạo một số "đối tượng trạng thái", đóng gói hành vi của đối tượng chính (gọi là Context) tương ứng với từng trạng thái (state-specific behavior). Khi trạng thái thay đổi, đối tượng chính sẽ ủy quyền thực hiện hành vi cho đối tượng trạng thái hiện hành.

Mẫu thiết kế State định nghĩa giao diện chứa các phương thức phụ thuộc trạng thái, mỗi cài đặt cho giao diện này sẽ định nghĩa một đối tượng trạng thái với phương thức phụ thuộc trạng thái thích hợp.

1. Cài đặt



- Context: đối tượng Client quan tâm, chứa trong nó:
 - + đối tượng trạng thái State, định nghĩa trạng thái hiện tại.
 - + phương thức request() là hành vi chỉ định bởi trạng thái hiện tại, request1() là hành vi không phụ thuộc trạng thái.
 - State: giao diện chung đóng gói hành vi tương ứng với một trạng thái của Context.
 - ConcreteState: các lớp cài đặt giao diện State, mỗi lớp cài đặt hành vi cụ thể tương ứng một trạng thái cụ thể của Context. Các hành vi này có thể thiết lập trạng thái mới, chuyển Context đến trạng thái kế tiếp.
- State và các ConcreteState thường được cài đặt như các lớp nội (inner class) của Context.
- Ví dụ, việc phân phối hàng lần lượt trải qua ba trạng thái: đang xử lý (Processing), đang chuyển (OnRoute) và đã đến người nhận (AtDestination). Hai phương thức phụ thuộc trạng thái là getCurrentLocation() (báo vị trí) và goNext() (chuyển sang trạng thái tiếp sau).



// (1) State

```
interface DeliveryState {
    void goNext(Delivery delivery);
    String getLocation();
}
```

// (2) ConcreteState

```
class Processing implements DeliveryState {
    @Override public void goNext(Delivery delivery) {
        delivery.setState(new OnRoute());
    }

    @Override public String getLocation() { return "Warehouse"; }
}

class OnRoute implements DeliveryState {
    @Override public void goNext(Delivery delivery) {
        delivery.setState(new AtDestination());
    }

    @Override public String getLocation() { return "On the train"; }
}
```

```

class AtDestination implements DeliveryState {
    @Override public void goNext(Delivery delivery) {
        delivery.setState(new AtDestination());
    }

    @Override public String getLocation() { return "Final destination"; }
}

// (3) Context
class Delivery {
    private DeliveryState state = new Processing();
    public Delivery() {
        System.out.println(getCurrentLocation());
    }

    public void setState(DeliveryState state) {
        this.state = state;
    }

    public Delivery goNext() {
        state.goNext(this);
        System.out.println(getCurrentLocation());
        return this;
    }

    private String getCurrentLocation() {
        return state.getLocation();
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("---- State Pattern ----");
        Delivery delivery = new Delivery();
        delivery.goNext().goNext();
    }
}

```

2. Liên quan

- Flyweight: giải thích tình huống và cách các đối tượng State được chia sẻ.
- Singleton: các đối tượng State thường là các Singleton để tránh tạo mới đối tượng State khi chuyển trạng thái.
- Strategy: State và Strategy có cùng cấu trúc tính (sơ đồ lớp) nhưng khác nhau về mục đích. Với State, client có ít hoặc không có kiến thức của các đối tượng State cụ thể. Context thường quyết định các đối tượng State trạng thái ban đầu và chuyển tiếp. Với Strategy, client thường nhận thức của các đối tượng Strategy khác nhau và chịu trách nhiệm cho việc khởi tạo Context cho một Strategy cụ thể.

3. Java API

Phương thức execute() của javax.faces.lifecycle.Lifecycle (điều khiển bởi FacesServlet, hành vi độc lập với phase (state) hiện hành của vòng đời JSF).

4. Sử dụng

Ta có:

- Các đối tượng sẽ thay đổi hành vi của chúng trong thời gian chạy, dựa trên một số ngữ cảnh.
- Các đối tượng đang trở nên phức tạp với nhiều nhánh điều kiện.

Ta muốn:

- Thay đổi tập xử lý cho các yêu cầu động đến một đối tượng.
- Giữ sự linh hoạt trong việc gán các yêu cầu để xử lý.

5. Bài tập

a) Khi ATM đang trong một trạng thái, người dùng có bốn tùy chọn ([insert card], [eject card], [insert PIN], [request cash]). Tùy theo trạng thái hiện hành, ATM sẽ có hành vi khác nhau.

Trạng thái NoCard: chưa đưa thẻ ATM vào máy.

[insert card]: → HasCard. Yêu cầu "Please enter a PIN".

[eject card]: báo "Enter a card first".

[insert PIN]: báo "Enter a card first".

[request cash]: báo "Enter a card first".

Trạng thái HasCard: đã đưa thẻ ATM vào máy.

[insert card]: báo "You can't enter more than one card".

[eject card]: báo "Card ejected", → NoCard.

[insert PIN]: nếu PIN đúng, báo "Correct PIN", → HasPin. Nếu PIN sai, báo "Incorrect PIN", đẩy card ra và báo "Card ejected", → NoCard.

[request cash]: báo "Enter PIN first"

Trạng thái HasPin: mã PIN nhập hợp lệ.

[insert card]: báo "You can't enter more than one card".

[eject card]: báo "Card ejected", → NoCard.

[insert PIN]: báo "Already entered PIN".

[request cash]: nếu số tiền yêu cầu (amount) lớn hơn tiền mặt (cash) hiện có trong máy, báo "D'ont have that cash", đẩy card ra và báo "Card ejected", → NoCard. Nếu số tiền yêu cầu hợp lệ ($\text{amount} \leq \text{cash}$), thanh toán và vẫn ở trạng thái HasPin; mỗi lần thanh toán, kiểm tra nếu tiền mặt đã hết ($\text{cash} = 0$), → NoCash.

Trạng thái NoCash:

[insert card]: báo "We don't have money".

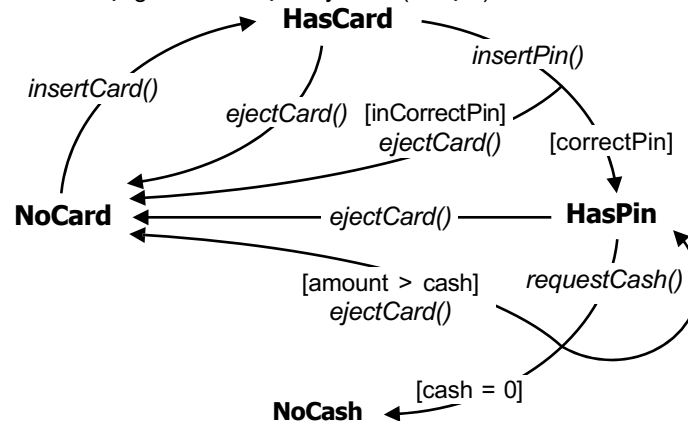
[eject card]: báo "We don't have money".

[insert PIN]: báo "We don't have money".

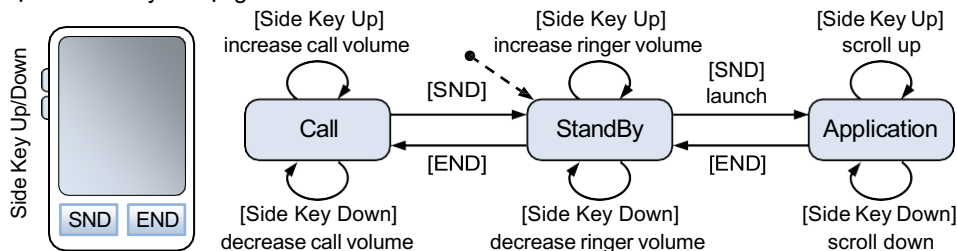
[request cash]: báo "We don't have money".

Hãy áp dụng mẫu thiết kế State để thay đổi hành vi của máy ATM khi trạng thái của nó thay đổi.

Sơ đồ chuyển trạng thái sau mô tả bốn trạng thái của một máy ATM (tô đậm).



b) Thiết kế một cell phone. Do kích thước giới hạn nên chỉ có 4 phím: 2 phím bề mặt là SND và END, 2 phím trên cạnh là Side Key Up và Side Key Down. Ngoài ra, cùng một phím có thể ánh xạ những chức năng khác nhau tùy theo chế độ (trạng thái) hiện tại của thiết bị. Sơ đồ chuyển trạng thái như sau:



Chế độ StandBy: thiết bị đang trong chế độ chờ.

[SND]: chuyển qua chế độ Call.

[END]: chuyển sang chế độ Application.

[Side Key Up/Down]: tăng/giảm âm lượng chuông.

Chế độ Call: thiết bị đang trong chế độ thoại.

[SND]: báo lỗi.

[END]: chuyển sang chế độ StandBy.

[Side Key Up/Down]: tăng/giảm âm lượng thoại.

Chế độ Application: thiết bị đang trong chế độ chạy ứng dụng.

[SND]: báo lỗi.

[END]: chuyển sang chế độ StandBy.

[Side Key Up/Down]: cuộn màn hình lên/xuống.


Hãy áp dụng mẫu thiết kế State để thay đổi hành vi của cell phone khi chế độ của nó thay đổi.

Hướng dẫn:

- Cài đặt mặc định cho phương thức [SND] và [END] là báo lỗi.

- Các phương thức cho [SND] và [END] nhận tham số là đối tượng CellPhone để có thể gọi phương thức thay đổi chế độ của nó.

Strategy

Encapsulate an algorithm 

Phần dễ thay đổi nhất trong chương trình là thuật toán, trong chương trình đôi khi bạn cần lựa chọn một trong nhiều thuật toán để giải quyết vấn đề. Ví dụ, để sắp xếp một danh sách, bạn có thể lựa chọn nhiều "chiến lược" sắp xếp: theo tên, theo thời điểm thay đổi, theo kích thước, .v.v...

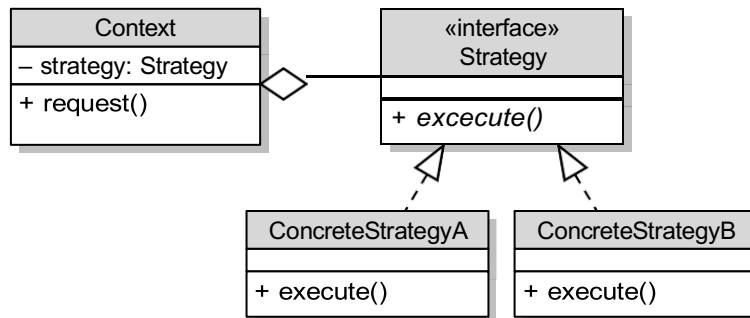
Mẫu thiết kế Strategy tách code thực thi thuật toán ra, đóng gói chúng, tạo thành một họ các thuật toán. Điều này cho phép chương trình có khả năng thay đổi thuật toán động trong thời gian chạy.

Ý tưởng là tách phần xử lý thuật toán ra khỏi đối tượng, từ đó cho phép thay đổi thuật toán độc lập với đối tượng dùng nó.

Ta tạo một số đối tượng thể hiện thuật toán và cho phép một đối tượng Context thay đổi đối tượng thuật toán của nó (thay đổi "chiến lược"), đối tượng thuật toán được áp dụng này sẽ thực hiện thuật toán cụ thể cho Context.

Ta dùng mẫu thiết kế này khi muốn tạo một đối tượng (Context) hỗ trợ linh hoạt nhiều thuật toán cùng một họ.

1. Cài đặt



- Strategy: giao diện cho họ thuật toán, đóng gói hành vi cho thuật toán. Context sử dụng giao diện này để thực hiện thuật toán cụ thể được cài đặt bởi một ConcreteStrategy.

- ConcreteStrategy: cài đặt một thuật toán cụ thể, sử dụng giao diện Strategy.

- Context: lớp áp dụng nhiều biến thể khác nhau của thuật toán. Đây là lớp sử dụng thuật toán thông qua giao diện Strategy.

Trong lớp này có thể định nghĩa một giao diện giúp Strategy truy cập được dữ liệu của nó.

Cách sử dụng các đối tượng Strategy của Context theo nguyên tắc Dependency Injection, đối tượng phụ thuộc không phải tạo trước mà được "tiêm" vào khi cần.

```

// (1) Context
class Context {
    private Strategy strategy;
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void request(String s) {
        strategy.algorithm(s);
    }
}

// (2) Strategy
interface Strategy {
    void algorithm(String s);
}

// (3) ConcreteStrategy
class UpperStrategy implements Strategy {
    @Override public void algorithm(String s) {
        System.out.println(s.toUpperCase());
    }
}

class LowerStrategy implements Strategy {
    @Override public void algorithm(String s) {
        System.out.println(s.toLowerCase());
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Strategy Pattern ---");
        Strategy upperStrategy = new UpperStrategy();
        Strategy lowerStrategy = new LowerStrategy();
        Context context = new Context();
        context.setStrategy(upperStrategy);
        context.request("Design Patterns");
    }
}
  
```

```

    context.setStrategy(lowerStrategy);
    context.request("Design Patterns");
}
}

```

Context gọi request() với các đối tượng Strategy khác nhau, được "tiêm" vào nó. Tùy theo loại đối tượng Strategy được áp dụng, request() chạy các thuật toán tương ứng, đóng gói trong đối tượng đó.

2. Liên quan

- Bridge: sử dụng một đối tượng để thực hiện tác vụ thực tế của nó.
- Flyweight: các đối tượng Strategy tạo từ các đối tượng Flyweight sẽ tốt hơn.
- State: State và Strategy có cùng cấu trúc tĩnh (sơ đồ lớp) nhưng khác nhau về mục đích. Xem mẫu thiết kế State.

3. Java API

Khi tạo GUI với một layout cụ thể, ta cung cấp một đối tượng LayoutManager cho container. Khi container cần thực hiện thuật toán bố trí các component ("chiến lược" layout), nó gọi phương thức của đối tượng LayoutManager tương ứng.

Trong trường hợp này: Context là Container, Strategy là LayoutManager, ConcreteStrategy là loại LayoutManager cụ thể (BorderLayout, GridLayout).

Một ví dụ khác là đối tượng Comparator được truyền đến phương thức sort() của Collection. Đối tượng này đóng gói thuật toán so sánh cụ thể:

```

import java.util.Collections;
import java.util.Comparator;
// ...

```

```

Comparator comparator = new BookComparatorByName();
Collections.sort(books, comparator);

```

Trong đó: Context là Collections, Strategy là Comparator, ConcreteStrategy là loại Comparator cụ thể (BookComparatorByName).

4. Sử dụng

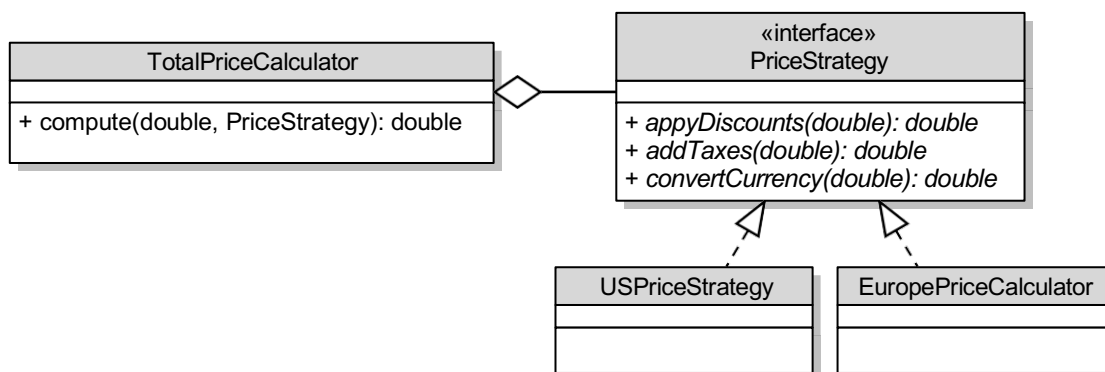
Ta có:

- Nhiều lớp liên quan nhau và chỉ khác nhau ở hành vi của chúng. Strategy cung cấp cách cấu hình một lớp có nhiều hành vi.
- Cần các biến thể khác nhau của thuật toán với mục đích nhất định.
- Các thuật toán dùng dữ liệu mà Client chưa biết đến. Dùng Strategy để tránh bộc lộ cấu trúc dữ liệu của thuật toán.
- Context định nghĩa nhiều hành vi, chúng xuất hiện theo các nhánh điều kiện, chuyển các hành vi này vào các Strategy.

5. Bài tập

a) Một trò chơi cho phép thiết lập nhiều cấp độ chơi: EASY, MEDIUM, HARD. Cấu hình cho độ khó của trò chơi được đóng gói trong DifficultyStrategy, bao gồm những hạn chế của người chơi, số lượng và độ khó của các nhiệm vụ con phải hoàn thành, ... Tùy lựa chọn ban đầu của người chơi, chiến lược thích hợp sẽ được áp dụng. Bạn hãy áp dụng mẫu thiết kế Strategy để thực hiện yêu cầu này.

b) Có hai cách tính giá tiền khác nhau với loại ngoại tệ được sử dụng, đều trả về VND. Hai cách này khác nhau về chế độ giảm giá (discount), thuế suất (tax) và cách chuyển đổi sang VND. Bạn hãy áp dụng mẫu thiết kế Strategy để thực hiện yêu cầu này.



Template Method

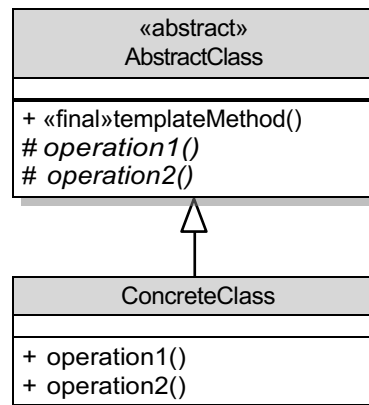
Algorithm skeleton 

Mẫu thiết kế Template Method cung cấp khung sườn của một hành vi/thuật toán, được hình thành bởi một *chuỗi thao tác* có thứ tự hoặc không cần thứ tự. Các lớp con trong mẫu thiết kế này có thể định nghĩa lại các thao tác trong chuỗi thao tác, độc lập với cấu trúc khung sườn của hành vi/thuật toán đó. Khả năng này làm cho việc thực hiện hành vi/thuật toán trở nên linh động.

Ngoài ra, mẫu thiết kế này cho phép chèn thêm một số phương thức "móc" (hook) vào chuỗi thao tác tại một số điểm đặc biệt, cho phép mở rộng hành vi/thuật toán tại các điểm đó.

Mẫu thiết kế Template Method rất phù hợp khi tạo ra các framework (khung công việc), trong đó ta cung cấp một thuật toán (một kiểu framework) linh hoạt để thực hiện một công việc. Các đơn thể phần mềm với vòng đời (life cycle) bao gồm các phương thức callbacks, cũng được cài đặt theo mẫu thiết kế Template Method.

1. Cài đặt



- AbstractClass: chứa
 - + Một phương thức cho thuật toán (templateMethod()), phương thức này thường là final để không bị viết lại (overridden).
 - + Các phương thức trừu tượng cho các thao tác cơ bản tạo nên thuật toán. Có thể định nghĩa các thao tác cơ bản này để cung cấp hành vi mặc định. Phương thức thuật toán gọi các thao tác cơ bản này theo thứ tự phù hợp.
- ConcreteClass: lớp dẫn xuất từ AbstractClass, cài đặt cụ thể và tách biệt cho các thao tác cơ bản của thuật toán. Nói cách khác, mỗi ConcreteClass cài đặt cho một biến thể của thuật toán.

Ý tưởng là lớp cơ sở (AbstractClass) khai báo các placeholders (chỗ đặt trước) cho một thuật toán, lớp dẫn xuất (ConcreteClass) sẽ lựa chọn hiện thực cho các placeholders này.

Thường có ba kiểu tác vụ khác nhau được gọi từ templateMethod():

```

abstract class AbstractClass {
    public final void templateMethod() {
        operation1();
        operation2();
        operation3();
    }
    protected abstract void operation1();
    protected void operation2() { }
    protected final void operation3() { }
}
  
```

- + operation1(): tác vụ trừu tượng được khai báo (và dùng) trong lớp cơ sở; lớp con sẽ định nghĩa cụ thể nó.
- + operation2(): tác vụ được định nghĩa trong lớp cơ sở, tác vụ này có thể rỗng hoặc được cài đặt mặc định. Tùy nhu cầu mở rộng thuật toán, lớp con có thể định nghĩa lại hoặc không cần định nghĩa lại tác vụ này. Chúng gọi là các phương thức "hook", do lớp con có thể "móc" vào thuật toán tại một số điểm.
- + operation3(): tác vụ không thay đổi, định nghĩa bước bắt buộc phải có trong thuật toán.

```
import java.util.regex.Pattern;
```

```

class Message {
    String address;
    String subject;
    String content;

    public Message(String address, String subject, String content) {
        this.address = address;
        this.subject = subject;
        this.content = content;
    }
}
  
```

```

// (1) AbstractClass
abstract class MessageSender {
    protected Message message;
    public final void execute(Message message) {
  
```

```

        this.message = message;
        initialize().sendMessage().cleanUp();
    }
    protected abstract MessageSender initialize();
    protected abstract MessageSender sendMessage();
    protected abstract MessageSender cleanUp();
}

// (2) ConcreteClass
class EmailMessageSender extends MessageSender {
    private boolean status = false;
    private String log = "Send email message failed";
    private boolean isEmail(String address) {
        return Pattern.compile("^([A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,6})$",
            Pattern.CASE_INSENSITIVE).matcher(address).find();
    }

    @Override protected MessageSender initialize() {
        status = isEmail(message.address);
        return this;
    }

    @Override protected MessageSender sendMessage() {
        if (status) {
            System.out.println("Sending by email...");
            log = "Send message to " + message.address + " successful";
        }
        return this;
    }

    @Override protected MessageSender cleanUp() {
        status = false;
        System.out.println("[LOG]: " + log);
        return this;
    }
}

class HttpPostMessageSender extends MessageSender {
    private boolean status = false;
    private String log = "Send HTTP message failed";
    private boolean isURL(String address) {
        return Pattern.compile("\\b(https?|ftp|file):/[\\-a-zA-Z0-9+&@#/%?~_!|:,;]*[\\-a-zA-Z0-9+&@#/%~_]",
            Pattern.CASE_INSENSITIVE).matcher(address).find();
    }

    @Override protected MessageSender initialize() {
        status = isURL(message.address);
        return this;
    }

    @Override protected MessageSender sendMessage() {
        if (status) {
            System.out.println("Sending by HTTP post...");
            log = "Send message to " + message.address + " successful";
        }
        return this;
    }

    @Override protected MessageSender cleanUp() {
        status = false;
        System.out.println("[LOG]: " + log);
        return this;
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("--- Template Method Pattern ---");
        Message eMessage = new Message("billgates@microsoft.com", "to Bill", "Hello Bill!");
        Message wMessage = new Message("http://oracle.com/forum", "to James", "Hello James!");
        new EmailMessageSender().execute(eMessage);
    }
}

```



```

    new HttpPostMessageSender().execute(wMessage);
}
}

```

Quy trình gửi một thông điệp, bằng email hoặc bằng web, phải qua một số bước: khởi tạo (initialize), gửi (sendMessage) và dọn dẹp (cleanUp). Thao tác cho các bước này được cài đặt độc lập với quy trình gửi thông điệp, phù hợp với loại thông điệp.

2. Liên quan

- Factory Method: một trong các bước của Template Method thường là tạo đối tượng, khi đó nó dùng Factory Method.
- Strategy: Template Method dùng thừa kế để thay đổi một phần của thuật toán, Strategy dùng ủy nhiệm để thay đổi hoàn toàn thuật toán.
- Observer: mẫu thiết kế Observer thường được dùng kết hợp với mẫu thiết kế Template Method như ví dụ sau.

```

abstract class ProcessManager extends Observable {
    protected final void process() {
        try {
            doProcess();
            setChanged();
            notifyObservers();
        } catch (Throwable t) {
            Log.error("ProcessManager.process(): ", t);
        }
    }
    abstract protected void doProcess();
}

```

Khi áp dụng mẫu thiết kế Template Method, sau khi doProcess() làm thay đổi dữ liệu thì các Observer đăng ký với ProcessManager sẽ được thông báo và thay đổi phần hiển thị tương ứng.

3. Java API

Tất cả các phương thức non-abstract của java.io.InputStream, java.io.OutputStream, java.io.Reader và java.io.Writer.

Tất cả các phương thức non-abstract của java.util.ArrayList, java.util.AbstractSet và java.util.AbstractMap.

4. Sử dụng

Ta muốn:

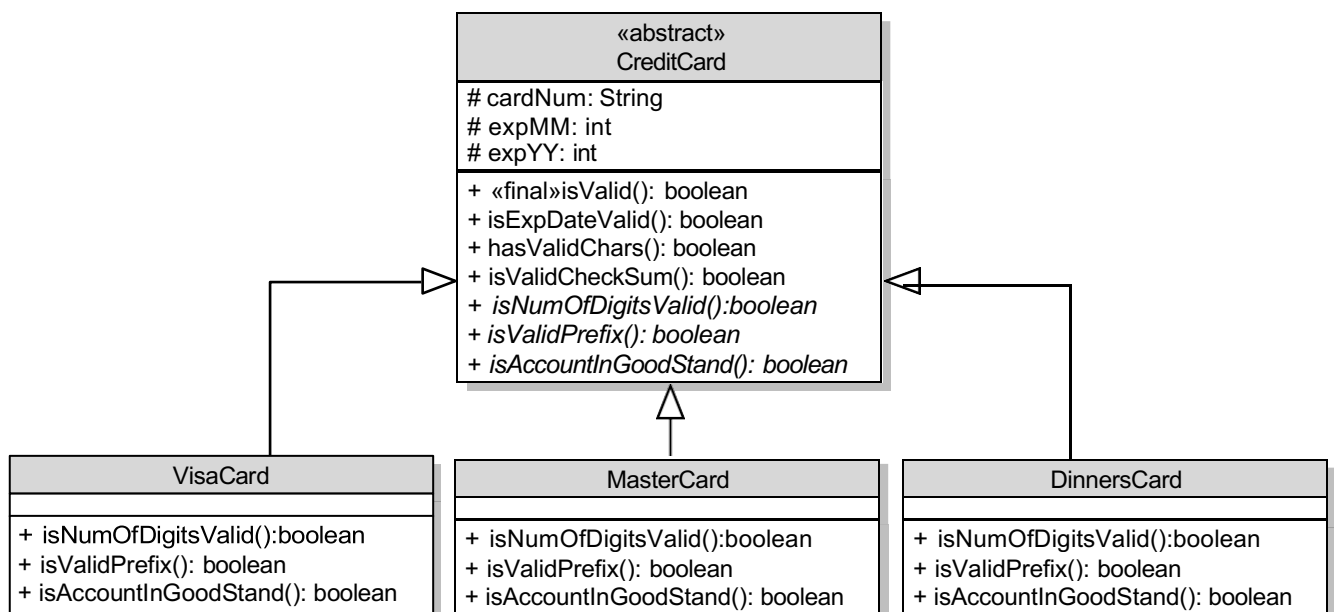
- Cài đặt những phần không thay đổi của một thuật toán trong một lớp đơn và những phần thay đổi của thuật toán trong các lớp dẫn xuất của lớp đơn đó.
- Hành vi chung của các lớp dẫn xuất được chuyển đến một lớp đơn duy nhất để tránh trùng lặp.

5. Bài tập

Quy trình kiểm tra tính hợp lệ của ba loại credit card (Visa, MasterCard, Dinners Club) đều có 6 bước, trong đó có một số bước giống nhau (1, 4, 5) và một số bước khác nhau (2, 3, 6).

Bước	Kiểm tra	Visa	MasterCard	Dinners Club
1	Hạn sử dụng (Expiration date)	> today	> today	> today
2	Chiều dài dãy số (Length)	13, 16	16	14
3	Dãy số đầu (Prefix)	4	51 – 55	30, 36, 38
4	Ký tự hợp lệ (Valid characters)	0 – 9	0 – 9	0 – 9
5	Thuật toán kiểm tra (Check digit algorithm)	Thuật toán Luhn	Thuật toán Luhn	Thuật toán Luhn
6	Trạng thái tài khoản (Account in good standing)	Visa API	MasterCard API	Dinners Club API

Hãy áp dụng mẫu thiết kế Template Method cho việc kiểm tra tính hợp lệ cả ba loại card trên.



Thuật toán kiểm tra (Check digit algorithm) gọi là thuật toán Luhn:

- Duyệt dãy số từ phải sang trái, số tại vị trí có thứ tự chẵn thì nhân đôi.

1	9	4	7	7	4	9	1	5
1	9x2	4	7x2	7	4x2	9	1x2	5
1	18	4	14	7	8	9	2	5

- Nếu kết quả nhân đôi này có hai chữ số thì lấy tổng hai chữ số đó làm kết quả cuối.

1	1+8	4	1+4	7	8	9	2	5
1	9	4	5	7	8	9	2	5

- Cộng các số cuối cùng này lại với nhau. Nếu kết quả chia hết cho 10 thì dãy số kiểm tra là hợp lệ.

$1 + 9 + 4 + 5 + 7 + 8 + 9 + 2 + 5 = 50 \rightarrow \text{valid}$

```
public boolean isLuhn(String cardNum) {
    int sum = 0;
    for (int i = cardNum.length() - 1; i >= 0; i -= 2)
        sum += cardNum.charAt(i) - '0';
    for (int i = cardNum.length() - 2; i >= 0; i -= 2)
        sum += 2 * (cardNum.charAt(i) - '0') % 9;
    return (sum % 10 == 0);
}
```

Visitor

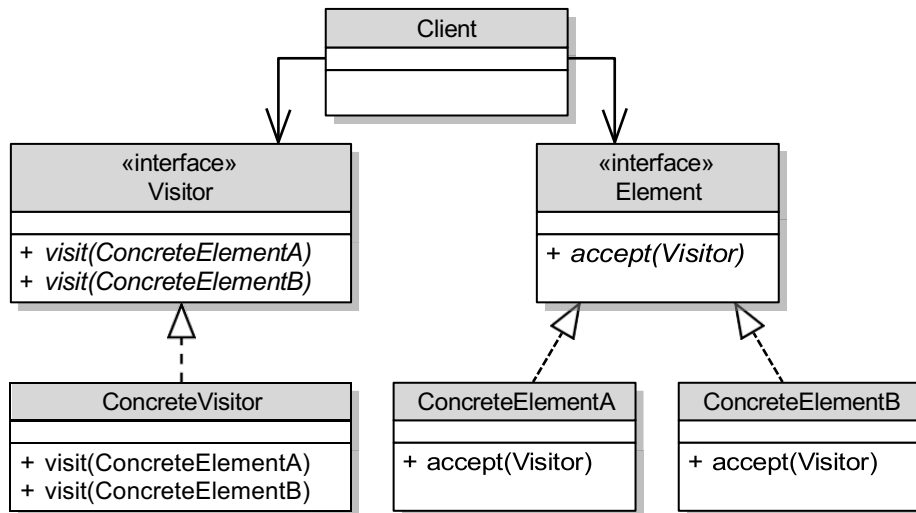
Visit tree node 

Một cách để thực hiện một tác vụ lên các đối tượng khác nhau trong một cấu trúc phức hợp là cung cấp thao tác đó riêng cho từng lớp của chúng. Thay vì tiếp cận từ bên ngoài vào như trên, có thể đảo ngược cách tiếp cận: gửi một đối tượng vào trong cấu trúc phức hợp và để cho đối tượng đó làm việc, mà vẫn không làm thay các lớp của cấu trúc phức hợp.

Mẫu thiết kế Visitor đóng gói tất cả những thao tác cần thiết cho tác vụ "viếng thăm" (print, upgrade, render, display, ...) vào một lớp riêng. Đối tượng thuộc lớp này, gọi là Visitor, chứa tác vụ "viếng thăm" phù hợp với lớp đối tượng tiếp nhận nó.

Các đối tượng Element, tức các đối tượng sẽ được "viếng thăm" trong cấu trúc phức hợp, phải chấp nhận "tiêm" đối tượng Visitor vào chúng, bằng phương thức accept(), để đối tượng Visitor đó có thể thực hiện tác vụ "viếng thăm" trên nó được. Như vậy, cách sử dụng các đối tượng Visitor tuân theo nguyên tắc Dependency Injection.

1. Cài đặt



- Visitor: định nghĩa giao diện chứa các phương thức visit() cho từng lớp ConcreteElement trong cấu trúc phức hợp. Tùy loại Element, Visitor sẽ "viếng thăm" bằng phương thức visit() tương ứng.

- ConcreteVisitor: cài đặt các tác vụ khai báo trong Visitor, thể hiện thuật toán "viếng thăm" cụ thể: đếm số Element, hiển thị nội dung Element, tính toán tích lũy, ... số các tác vụ này có thể mở rộng.

- Element: khai báo giao diện chung cho Element, quan trọng là phương thức accept(), nhận Visitor như đối số. Chú ý, chính phương thức accept() này sẽ gọi các phương thức visit() của Visitor mà nó chấp nhận, được truyền đến nó như đối số.

- ConcreteElement: loại Element cụ thể tạo nên cấu trúc phức tạp. Số ConcreteElement là cố định.

```
import java.util.ArrayList;
import java.util.Arrays;
```

```
// (1) Element
interface FileSystemNode {
    void accept(Visitor visitor);
}
```

```
// (2) ConcreteElement
class FileNode implements FileSystemNode {
    private String name;
    public FileNode(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

```
class FolderNode implements FileSystemNode {
    private String name;
    ArrayList<FileSystemNode> list = new ArrayList<>();
    public FolderNode(String name, FileSystemNode... children) {
        this.name = name;
        this.list.addAll(Arrays.asList(children));
    }
}
```

```

    @Override public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public FolderNode add(FileSystemNode node) {
        list.add(node);
        return this;
    }

    public String getName() {
        return name;
    }
}

// (3) Visitor
interface Visitor {
    void visit(FileNode element);
    void visit(FolderNode element);
}

// (4) ConcreteVisitor
class PrintVisitor implements Visitor {
    @Override public void visit(FileNode node) {
        System.out.println(node.getName());
    }

    @Override public void visit(FolderNode node) {
        System.out.println "[" + node.getName() + "]";
        for (FileSystemNode e : node.list) {
            if (e instanceof FileNode) visit((FileNode)e);
            else if (e instanceof FolderNode) visit((FolderNode)e);
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println("---- Visitor Pattern ----");
        FolderNode root =
            new FolderNode("java",
                new FileNode("readme.txt"),
                new FolderNode("javaSE",
                    new FileNode("code.java"),
                    new FolderNode("tutorial")),
                new FolderNode("javaEE",
                    new FileNode("web.pdf"),
                    new FileNode("ejb.pdf")));
        root.accept(new PrintVisitor());
    }
}

```

Chú ý constructor của FolderNode, phương thức này cho phép tạo thành một cấu trúc phức hợp tương tự cây thư mục, bao gồm FileNode và FolderNode; FolderNode có thể chứa các FileNode và FolderNode khác. Client chọn một PrintVisitor cài đặt các phương thức visit(), hiển thị thông tin tùy FileNode hoặc FolderNode, trong đó phương thức visit() của FolderNode là đệ quy. Client gửi PrintVisitor này đến cấu trúc phức hợp bằng cách truyền PrintVisitor cho phương thức accept().

2. Liên quan

- Composite: cấu trúc đối tượng phức hợp mà Visitor áp dụng tác vụ lên đó, thường được định nghĩa bởi Composite.
- Interpreter: Visitor cũng có thể hỗ trợ Interpreter thực hiện tác vụ diễn dịch của nó.

3. Java API

javax.lang.model.element.AnnotationValue và AnnotationValueVisitor.

javax.lang.model.element.Element và ElementVisitor.

javax.lang.model.type.TypeMirror và TypeVisitor.

4. Sử dụng

Ta muốn:

- Duyệt qua một hệ thống phân lớp phức tạp được bảo vệ chặt chẽ, khó thay đổi.
- Một hệ thống cần nhiều tác vụ khác nhau thực hiện trên nó, có thể mở rộng số tác vụ này. Dùng như một parser.
- Các tác vụ gắn liền với các loại đối tượng trong hệ thống phân cấp.

5. Bài tập

Cung cấp một Visitor truy cập hệ thống các tập tin và thư mục từ một vị trí chỉ định, in ra tên của các tập tin và thư mục so trùng với một mẫu biểu thức chính quy (regular expression) chỉ định. Giải quyết vấn đề nếu hệ thống hỗ trợ cả link. Link được dùng trên Unix, tương tự shortcut trên Windows, là một bản sao ảo của tập tin hoặc thư mục, chứa tên link và đường dẫn của tập tin hoặc thư mục mà nó đại diện. Link có các tác vụ giống như tập tin và thư mục, ngoại trừ việc ta có thể xóa nó mà không ảnh hưởng đến tập tin hoặc thư mục mà nó đại diện.

Hướng dẫn: áp dụng mẫu thiết kế Proxy cho link.

Tài liệu tham khảo

(Theo năm xuất bản)

- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – **Design Patterns: Elements of Reusable Object-Oriented Software** – Addison-Wesley, 1995. ISBN 0201633612
- [2] Mark Grand – **Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition** – John Wiley & Sons, 2002. ISBN 0471227293
- [3] Partha Kuchana – **Software Architecture Design Patterns in Java** – Auerbach Publications, 2004. ISBN 0849321425
- [4] Allen Holub – **Holub on Patterns: Learning Design Patterns by Looking at Code** – Apress, 2004. ISBN 978-1590593882
- [5] Timothy C. Lethbridge, Robert Laganière – **Object-Oriented Software Engineering** – McGraw-Hill, 2005. ISBN 0077097610
- [6] Cay Horstmann – **Object-Oriented Design and Patterns, Second Edition** – John Wiley & Sons, 2006. ISBN 978-0471744870
- [7] Steven John Metsker, William C. Wake – **Design Patterns in Java™** – Addison-Wesley, 2006. ISBN 0321333020
- [8] Paul R. Allen, Joseph J. Bambara – **SCEA Sun® Certified Enterprise Architect for Java™ EE Study Guide (Exam 310-051)** – McGraw-Hill, 2007. ISBN 0071510931
- [9] Christopher G. Lasater – **Design Patterns** – Wordware Publishing, 2007. ISBN 1598220314
- [10] Judith Bishop – **C# 3.0 Design Patterns** – O'Reilly, 2008. ISBN 978-0596527730
- [11] Joshua Bloch – **Effective Java™, Second Edition** – Addison-Wesley, 2008. ISBN 978-0321356683
- [12] Dale Skrien – **Object-Oriented Design using Java** – McGraw-Hill, 2009. ISBN 978-0072974164
- [13] Eddie Burris – **Programming in the Large with Design Pattern** – Pretty Print Press, 2012. ISBN 978-0072974164
- [14] Vaskaran Sarcar – **Java Design Patterns** – Apress, 2016. ISBN 978-1-4842-1802-0